Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Extraction automatisée d'un feature model à partir d'un catalogue de produits

Vanbeneden, Charles

Award date:
2011

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. May. 2023

Facultés Universitaires Notre-Dame de la Paix, Namur
Faculté d'informatique.
Année académique 2010-2011

Extraction automatisée d'un Feature Model à partir d'un catalogue de produits

Charles Vanbeneden

Mémoire présenté en vue de l'obtention du grade de master en informatique.

Résumé

L'utilisation de Feature Models (FMs) pour décrire les features communes et variables d'une ligne de produits logiciel est devenu un scénario ordinaire. Malheureusement, la tâche de création du FM à partir d'un catalogue de produit existant est à la fois sujette à erreurs, consommatrice en temps et nécessite un effort de modélisation important de l'utilisateur. C'est pourquoi nous présentons, dans ce mémoire, un processus d'extraction de FMs à partir de catalogues de produits (par exemple, documenté par des fichiers au format CSV ou Comma-Separated Values). Nous garantissons que le FM résultant représente de manière compacte les combinaisons valides des features supportées par les produits et comporte une hiérarchie de features lisible associée à l'information de variabilité. Un gestionnaire de ligne de produits peut paramétriser le processus d'extraction à partir de directives de haut niveau (par exemple, permettant de filtrer des produits ou caractéristiques). Notre approche repose sur une implémentation et nos résultats d'évaluation démontrent la montée en charge et la praticabilité de celle-ci pour différents ensembles de données publiques.

Abstract

The use of feature models to describe the common and variable features of software product lines (SPL) is becoming commonplace. Unfortunately, the task of creating a feature model (FM) for an existing source is both daunting and time-consuming, requiring substantial effort from a modeller. We present an automated process for extracting feature models from products catalogue (e.g., documented in CSV files - Comma-Separated Values). We guarantee that the resulting feature model compactly represents the valid combination of features supported by products and has a readable tree hierarchy together with variability information. An SPL practitioner can parametrize the extractive procedure through high-level directives (e.g., products/features scoping). The process is tool supported and our evaluation results demonstrate the scalability and practicability of the approach on various public data.

Avant-propos

Je voudrais remercier toutes les personnes qui m'ont aidé dans la réalisation de ce mémoire. J'aimerais remercier les personnes suivantes :

- Prof. Patrick Heymans, pour m'avoir donné l'opportunité de diriger mon mémoire dans un domaine de recherche intéressant, orienté vers un stage me correspondant vraiment et pour sa disponibilité.
- Prof. Philippe Lahire, pour m'avoir accueilli à l'Université de Nice-Sophia Antipolis, donné la ligne directrice de mes recherches et mis tout en œuvre pour le bon déroulement de mon stage.
- Mathieu Acher, pour sa disponibilité, ses conseils avisés, sa détermination, le bon travail collaboratif et sa sympathie.
- Philippe Collet, pour ses bons conseils au sein de l'équipe lors de mon stage et sa bonne humeur.
- Quentin Boucher, pour sa disponibilité et pour avoir relu successivement et de manière attentive mon mémoire lors de son élaboration.
- Andreas Classen, pour le support fourni lors du travail sur TVL.

J'aimerais aussi remercier ma famille et amis, et surtout Jonathan Bourgeois, pour m'avoir supporté et encouragé lorsque j'étais à l'étranger. J'adresse un remerciement tout particulier à mes parents qui n'ont eu de cesse de croire en moi et de m'épauler en toutes circonstances.

Table des matières

Chapitre 1

Introduction

Les Feature Models (FMs) sont couramment utilisés dans l'ingénierie des lignes de produits logiciels (Software Product Lines - SPLs) pour documenter et raisonner sur la variabilité relative à une famille de produits. Les produits sont alors décrits en terme de caractéristiques (appelées features), structurées hiérarchiquement, qui peuvent être obligatoires ou optionnelles, représenter des alternatives, ou dépendre d'autres caractéristiques. L'utilisation de FMs pour décrire les features communes et variables d'une SPL est devenu un scénario ordinaire.

La tâche de création du FM à partir d'un catalogue de produits existant est à la fois sujette à erreurs, consommatrice en temps et nécessite un effort de modélisation important de l'utilisateur. Dans la littérature, certains cas d'utilisation font état de la création d'un extracteur automatique de FM. Par exemple [12], concerne l'extraction de la variabilité autour de la configuration du noyau Linux pour déterminer s'il existe des features non sélectionnables (appelées features mortes). Ou encore, [36] relate la création d'un FM, contenant les caractéristiques d'un ensemble de logiciels antivirus, à partir de données disponibles sur un site web spécialisé. Chacune des features associées à ces logiciels provient de listes à puces HTML issues des pages de produits correspondants. De la même manière, nous avons remarqué que dans l'industrie, la variabilité des produits est déjà documentée dans des catalogues qui consistent souvent en des fichiers peu structurés caractérisant les produits selon différentes perspectives. Beaucoup d'entre-elles n'ont jamais utilisé les FMs à cause de l'effort important à apporter pour les créer.

Les 3 cas présentés au paragraphe précédent sont des illustrations de l'importance d'un processus d'extraction de FM à partir de données peu structurées. C'est pourquoi nous présentons, dans ce mémoire, un processus qui permet d'extraire de façon générique et semi-automatique un FM à partir d'un catalogue de produit existant. Le FM généré peut alors servir de point

de départ à des opérations de raisonnements ou pour créer un configurateur. De plus, nous garantissons que le FM résultant représente de manière compacte les combinaisons valides des features supportées par les produits et comporte une hiérarchie de features lisible associée à l'information de variabilité. Nous donnons des facilités au gestionnaire de ligne de produits pour paramétriser le processus d'extraction à partir de directives de haut niveau (par exemple, permettant de filtrer des produits ou caractéristiques). Notre approche repose sur une implémentation et nos résultats d'évaluation démontrent la montée en charge et la praticabilité de celle-ci pour différents ensembles de données publiques.

Notre travail est divisé comme suit :

Chapitre 2 dresse un bref état de l'art permettant de comprendre les diverses contributions associées à ce travail.

Chapitre 3 présente un cas d'utilisation qui servira à illustrer les différents aspects et exemples du processus d'extraction dans le reste de ce mémoire.

Chapitre 4 donne la motivation et la description du processus d'extraction complet.

Chapitre 5 donne une liste d'extensions au processus d'extraction ainsi qu'au formalisme de modélisation de FM associé.

Chapitre 6 décrit l'architecture générale et les choix d'implémentation.

Chapitre 7 donne un cas d'utilisation de bout en bout pour que l'utilisateur puisse comprendre comment utiliser notre processus d'extraction avec ses propres données.

Chapitre 8 évalue notre processus d'extraction en terme de performance et de validité des modèles générés. Des optimisations y sont définies pour permettre une meilleure montée en charge.

Chapitre 9 fait référence à une série de travaux connexes à notre approche.

Chapitre 10 présente la conclusion de notre travail ainsi que des idées de futurs travaux potentiels sur le sujet et le travail restant à faire pour que notre approche soit employée à large échelle.

Chapitre 2

Les lignes de produits logiciels

Une ligne de produits logiciel (Software Product Line – SPL) est « un ensemble critique de systèmes logiciels qui partagent un ensemble de features communes qui est géré pour satisfaire les besoins spécifiques d'un segment particulier du marché ou d'une mission et qui sont développés autour d'un ensemble commun d'actifs principaux (core assets) d'une manière prescrite » [19]. Le but de l'ingénierie des lignes de produits logiciels (Software Product Line Engineering ou SPLE) est de produire une famille de variantes de programmes associées à un domaine d'une manière systématique [19]. Le développement d'une SPL commence avec l'analyse du domaine pour identifier les points communs et différences entre les éléments de la ligne de produits. Pour ce faire, usuellement, on décrit la variabilité d'une SPL en terme de caractéristiques (ou *features*) qui sont pertinentes en terme d'abstraction du domaine et sont typiquement des incréments dans les fonctionnalités du programme. Un *modèle de features* (Feature Model ou FM) est utilisé pour définir de manière compacte toutes les features dans une SPL et leurs combinaisons valides ; il consiste typiquement en un graphe ET/OU avec des contraintes propositionnelles.

Dans ce chapitre, nous présentons en détail différents concepts autour de l'ingénierie des SPLs. Deux tâches découlent de l'SPLE : (1) la *modélisation* de la SPL avec le formalisme des FMs, décrit dans la Section 2.1. Différents types de FMs y sont énoncés ; (2) le *raisonnement* et la manipulation de ce FM avec toute une série d'opérateurs, présentés à la Section 2.2 qui permet, par exemple, d'obtenir la liste des configurations valides. Pour finir, nous discutons à propos des différentes approches de programmes de gestion des FMs. De manière générale, en SPLE, le gestionnaire de SPL va modéliser la ligne de produits manuellement, souvent aidé de ces outils. Il existe des approches de type graphiques et textuelles qui ont des particularités exposées dans la Section 2.3. Les concepts présentés dans ce chapitre permettent à un non-initié aux FMs de comprendre les contributions présentées dans les

chapitres suivants de ce mémoire.

2.1 Les Feature Models

Une *feature* est une caractéristique visible et intelligible d'une ligne de produits [18, 7]. Les FMs structurent hiérarchiquement les features en des niveaux de détails croissants. Les FMs sont souvent représentés par des *feature diagrams* (FDs) soit par une syntaxe textuelle. Un FD consiste en une représentation graphique d'un FM, c'est un arbre Et/Ou dont les nœuds sont des features et dont les liens représentent les relations. Un exemple de FD est présenté à la Figure 2.1. Au moins une feature est comprise dans l'arbre, la *racine* ①. Une *configuration* est une instance d'un FM particulier et correspond à un ensemble de features *sélectionnées* ou *désélectionnées*. Une feature peut être sélectionnée seulement si sa feature parente a été sélectionnée quelle que soit la relation avec celle-ci. La racine est par défaut toujours sélectionnée. La validité d'une configuration est déterminée par la sémantique du FM. Des outils sont fournis à l'utilisateur pour manipuler les configurations de son FM (ceux-ci sont décrits à la Section 2.2.2). Un *produit* est équivalent à une configuration complète et valide où les features sélectionnées sont stipulées, les features omises sont implicitement désélectionnées et où les features sélectionnées ne violent aucune relation (ou contrainte). Une *feature morte* est une feature qui n'apparaît dans aucun produit, c'est-à-dire que par les contraintes, celle-ci n'est jamais sélectionnable par l'utilisateur. Un FM représente la variabilité relative aux produits d'une SPL en terme de features et de relations entre elles.

Chaque feature a une relation particulière avec ses features filles. Les relations de groupes permettent de lier l'ensemble des features filles par une contrainte. Nous en avons identifié 7 :

- *Relation Obligatoire* ③ : La feature fille apparaît dans tous les produits dans lesquels la feature parente apparaît.
- *Relation Optionnelle* ④ : La liberté est donnée à l'utilisateur de sélectionner ou non cette feature.
- *Groupe Et* ② : Toutes les features du groupe sont sélectionnées.
- *Groupe Alternatif*[1] ⑤ : Une et une seule feature du groupe peut être sélectionnée.
- *Groupe Ou* ⑥ : Au moins une des features du groupe est sélectionnée.
- *Groupe Mutex* ⑦ : Au plus une feature du groupe peut être sélectionnée. Contrairement au *groupe Alternatif*, la configuration reste valide

1. Dans la littérature, différents noms désignent ce même concept : XOR-group (exemple dans [15, 22]), One-of-group [26],... Nous avons toutefois choisi celui-là, car c'est de cette manière qu'il a été défini pour la première fois dans [43].

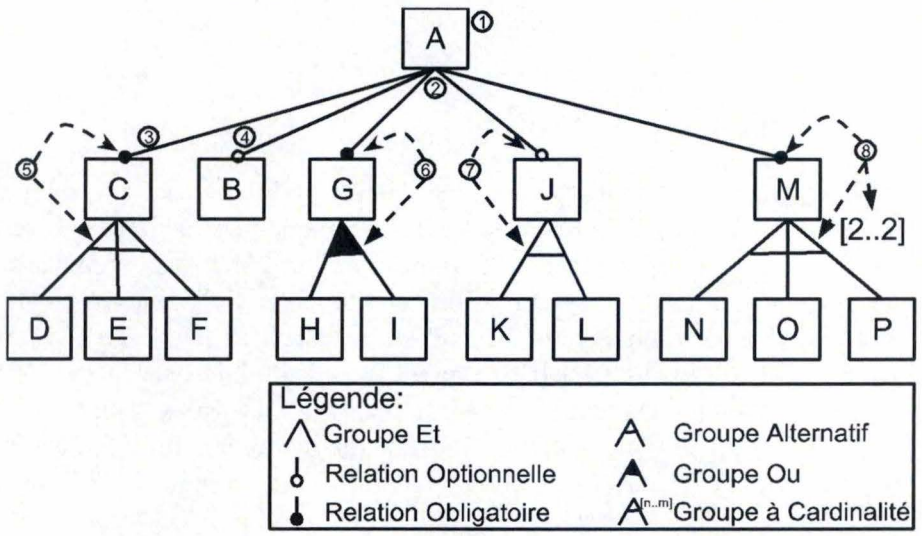

FIGURE 2.1 – Exemple de feature diagram reprenant toutes les constructions

si aucune feature du groupe n'est sélectionnée.

- *Groupe à Cardinalité* ⑧ : Une cardinalité de groupe correspond à un intervalle dénoté par `<n..m>`, avec $n \leq m$, qui limite le nombre de features filles qui peut faire partie d'un produit lorsque la feature parente est sélectionnée [11]. Cela implique que le *groupe Alternatif* correspond à un groupe à cardinalité `<1..1>`, le *groupe Ou* correspond à l'intervalle `<1..N>` où N est le nombre de features du groupe ainsi que le *groupe Et* à `<N..N>` et le *groupe Mutex* à `<0..1>`.

En plus de ces relations, il est possible d'ajouter explicitement des contraintes booléennes [2] sur les features :

- *Implication* : $A \rightarrow B$ signifie que si la feature A est sélectionnée, la feature B l'est aussi.
- *Négation* : $\neg A$ signifie que A ne peut pas être sélectionnée.
- *Ou* : $A \vee B$ signifie que au moins une des deux feature doit être sélectionnée.
- *Et* : $A \wedge B$ signifie que si A (respectivement B) est sélectionnée, alors B (respectivement A) l'est aussi. Et inversement, si A (respectivement B) est désélectionnée, alors B (respectivement A) est désélectionnée.
- Il est évidemment possible de combiner les 4 opérations booléennes de base présentées aux points précédents pour former des contraintes complexes.

2. Ces contraintes ne sont pas indiquées directement dans le feature diagram, mais y sont adjointes.

Il existe aussi des *contraintes implicites* qui sont des contraintes données par des structures particulières du FM considéré (dans cette section sont définis les *groupes Et*, *Ou*, *Alternatifs*, etc.). Ces structures peuvent être contournées en utilisant des *contraintes* dites *explicites*. Ce sont des formules propositionnelles qui sont données par l'utilisateur. Cependant, pour un FM d'une taille raisonnable, les formules booléennes deviennent conséquentes et illisibles pour le gestionnaire de la SPL. Les contraintes implicites sont donc utilisées autant que possible, car elles renforcent la sémantique de l'arbre et diminuent la complexité des contraintes explicites. Un FM bien modélisé est un FM contenant le plus d'information sémantique possible.

Dans la littérature [11], on distingue 3 types de FMs. Les *FMs de base* comprennent les relations *Obligatoire* et *Optionnelle*, les *groupes Et*, *Ou*, *Alternatif* et les contraintes booléennes. Les *FMs à Cardinalité* comprennent les mêmes constructions que les FMs de base, mais gèrent, en plus, les *groupes à cardinalité* particulière. Le troisième type de FM est le *FM à attributs*[3]. Il n'est pas présenté dans ce mémoire, car aucune implémentation de gestionnaire de FM ne les gère correctement. Le *groupe Mutex* ainsi que la *cardinalité de feature* sont des extensions propres à certaines implémentations de gestionnaire de FM.

2.2 Opérations sur les FMs

Des opérations de manipulation des FMs ont été définies pour répondre aux besoins en terme de gestion et de raisonnement du gestionnaire de la SPL. Nous allons énoncer les opérateurs les plus communément admis et implémentés. Il se pourrait donc qu'une opération ne soit pas incluse dans un gestionnaire de FM particulier ou que l'utilisateur doive fournir un effort supplémentaire pour les utiliser. En général, les implémentations de gestionnaire de FM sont des APIs dans des langages divers. Il est donc possible à l'utilisateur, si besoin est, de programmer les opérateurs décrits dans cette section avec le langage associé. En premier lieu, nous allons présenter l'environnement sous-tendant la manipulation de FMs. Ensuite, nous introduisons les opérations de configuration ainsi que les opérations de base qui sont disponibles dans presque toutes les implémentations de gestionnaire de FMs. Nous terminerons par définir l'opérateur de fusion de FMs qui est à la base de la contribution du Chapitre 4.

2.2.1 Environnement

Une fois l'activité de modélisation d'un FM finie, l'utilisateur obtient une version graphique et/ou textuelle de celui-ci. Suivant l'utilisation dans

3. Le concept d'attribut est défini dans [11, 10].

l'activité de raisonnement, l'utilisateur peut le charger autant de fois que nécessaire en mémoire. Il faut alors placer ces instances dans des entités identifiables, que nous allons appeler *variables*. La possibilité de placer les FMs dans des variables est une condition aux opérations décrites dans les sections suivantes. De la même manière, des boucles permettant d'itérer sur des ensembles de features, des configurations ou des FMs sont très intéressantes dans de multiples cas pratiques.

Des *accesseurs* existent pour permettre à l'utilisateur de naviguer dans le FM. En voici une liste non exhaustive :

- Retourner la feature racine (root).
- Retourner la liste des features enfant de l'actuelle (children).
- Retourner la feature parente de l'actuelle (parent).
- Retourner la liste des features sœurs de l'actuelle (sibling).
- Retourner le type de relation de la feature actuelle avec sa feature parente : *groupes Ou*, *Alternatif*, *Mutex*, *à Cardinalité* `<n..m>` et *relations Obligatoire* ou *Optionnelle* (operator).
- Retourner un nouveau FM à partir d'un sous-arbre du FM donné. Cette opération ne modifie pas le FM donné en entrée. Le FM produit aura les mêmes configurations que le sous-arbre associé (extract).

Pour permettre à l'utilisateur de travailler en groupe sur la même SPL ou FM, il peut être intéressant de lui donner la possibilité de *charger* un ensemble d'opérations à partir de fichiers. Cette facilité amène à la *réutilisation* d'opérateurs groupés afin de répondre à une tâche ou ayant une sémantique particulière.

Un opérateur très utilisé est l'opérateur d'*égalité*. Il permet de vérifier si deux FMs, configurations ou features sont identiques.

Tous les éléments que sont les variables, les opérateurs, les accesseurs et les structures de contrôle constituent l'environnement du gestionnaire de FM.

2.2.2 Configuration de FM

L'*autopropagation* est une opération qui prend un FM et une configuration partielle en entrée et retourne une nouvelle configuration avec les features qui devraient être sélectionnées et/ou supprimées en tant que résultat de la propagation des contraintes du modèle. Cette opération est implémentée par tous les gestionnaires de FM supportant la configuration, car elle est l'opération sine qua non de l'activité de configuration de FM. Cette activité consiste donc à utiliser les outils donnés par les différentes implémentations

de gestionnaire de FM.

Pour modifier une configuration [4], l'utilisateur peut utiliser les opérations suivantes :

- Sélectionner une feature : consiste à placer la feature donnée en paramètre dans la liste des features sélectionnées de la configuration (select).
- Désélectionner une feature : consiste à placer la feature donnée en paramètre dans la liste des features désélectionnées de la configuration (deselect).
- Annuler la sélection d'une feature : consiste à retirer la feature donnée en paramètre de la configuration (unselect).

Certaines facilités sont offertes à l'utilisateur pour accéder aux différentes features sélectionnées, désélectionnées et non encore configurées. Les opérations consistent en la génération d'une liste de ces différentes features qui est alors manipulable programmativement.

2.2.3 Opérations de base

Les opérations décrites dans cette section sont les plus communément admises et implémentées par les gestionnaires de FMs. Pour dresser la liste d'opérations suivante, nous nous sommes basé sur celles fournies par FAMILIAR, un gestionnaire de FM sur le marché.

Pour raisonner à propos des FMs, nous avons identifié 5 opérateurs :

- Vérifier si la configuration donnée en entrée est valide (isValid).
- Comparer deux FMs donnés et signaler si une relation de *refactoring*, *généralisation*, *spécialisation* ou une *modification arbitraire*, existe entre ceux-ci (compare). L'opérateur se base sur l'ensemble des configurations de chacun de ces FMs. Soit C_1 et C_2 l'ensemble des configurations respectivement du premier et du deuxième FM. Si $C_1 = C_2$, nous avons un *refactoring*, c'est-à-dire que les deux FMs ont peut être des hiérarchies différentes mais ils représentent le même ensemble de configurations. Si $C_1 \subset C_2$, alors on dit que C_2 *généralise* C_1 et que C_1 *spécialise* C_2. Si par contre aucun lien particulier n'est trouvé entre C_1 et C_2, on dit que les hiérarchies sont des *modifications arbitraires* l'une de l'autre.
- Compter le nombre de configurations valides (counting).

4. Un FM non encore configuré a les *core features* sélectionnées.

- Retourner le nombre de features faisant partie du FM donné en entrée (nbFeatures).
- Déterminer la liste des features mortes (deadFeaturesList).

Pour modifier les FMs, nous avons identifié 4 opérateurs :

- Permettre de changer le nom d'une feature (renameFeature). Le nom est bien entendu modifié dans les contraintes associées.
- Permettre de supprimer une feature (deleteFeature). Les contraintes associées à cette feature sont supprimées.
- Permettre de placer un FM donné en dessous d'une feature d'un autre FM (insert).
- Permettre d'ajouter, modifier ou supprimer des contraintes (alterConstraints).

Dans certains cas, l'ensemble des configurations valides (et donc des produits associés au FM) peut être modifié par l'utilisation de ces opérateurs.

2.2.4 Opérateur de fusion

Comme expliqué dans [5], lorsque différentes vues d'une SPL doivent être manipulées, il est plus que probable qu'elles partagent certaines features. Dans ce cas, l'opérateur de fusion peut être utilisé pour fusionner les parties des FMs qui se chevauchent et alors obtenir une version compacte de ceux-ci. Cet opérateur se base sur la correspondance des noms de features, c'est-à-dire que deux features sont des *pivots*[5] si elles ont le même nom.

Dans [2], les différentes approches de composition de FMs sont comparées. Il est déterminé qu'une implémentation basée sur la logique propositionnelle couplée à l'algorithme proposé dans [25] pour construire un FM à partir de la formule propositionnelle est efficient. En particulier, les approches concurrentes, la plupart basées sur des stratégies *syntaxiques* [55, 54, 1], ont des limitations pour représenter de manière précise l'ensemble des configurations voulu, et ce, surtout en la présence de contraintes interarbres. Une approche basée sur la logique propositionnelle a l'avantage de raisonner directement au niveau sémantique, en terme d'ensemble de configurations.

Comme présenté dans [3], l'idée générale est d'encoder chaque FM en entrée impliqué dans l'opération de fusion comme des formules propositionnelles[6], notées ϕ_{FM1}, ϕ_{FM2}, etc. En fonction de mode de fusion, on ap-

5. Un pivot est une feature qui sert de point de fusion.

6. L'ensemble des configurations représentées par un FM peut être décrit de manière compacte par une formule propositionnelle définie sur un ensemble de variables booléennes, où chacune d'elles correspondent à une feature [9, 25].

plique certaines opérations booléennes sur ces formules pour obtenir ϕ_{Result}, la formule booléenne résultant de la fusion. Finalement, l'algorithme présenté dans [25] transforme ϕ_{Result} en une hiérarchie de features, c'est-à-dire, un FM. Il construit un arbre avec des nœuds additionnels pour les groupes de features qui peuvent être traduites en un *FM basique*. En particulier, l'algorithme peut restaurer la hiérarchie des FMs donnée en entrée en indiquant les liens parent-enfants (différents groupes et relations).

Plusieurs modes sont définis pour l'opérateur de fusion en fonction de l'ensemble de configuration à conserver dans le FM résultant de la fusion. En voici quelque uns :

- L'*intersection* : est le mode le plus restrictif. En effet, il conserve les configurations communes de l'ensemble des FMs fusionnés.
- L'*union* : est le mode le moins restrictif. En effet, celui-ci garde les configurations valides de tous les FMs passés en entrée. Cela veut dire qu'une configuration valide d'un des FMs donne une configuration valide dans le FM généré.
- L'*union stricte* : regroupe toutes les configurations qui sont valides dans l'ensemble des FMs passés en entrée. Cela veut dire qu'une configuration valide du FM fusionné est valide dans un seul des FMs passés en entrée.
- La *différence* : conserve l'ensemble des configurations qui forme la différence entre les ensembles des configurations des FMs passés en entrée. C'est-à-dire, que pour deux FMs, les configurations gardées sont les configurations apparaissant dans l'un et pas dans l'autre et inversement.

D'autres modes existent, comme les *produits croisés*. Ils ne sont pas abordé car leur utilisation ne rentre pas dans le cadre de ce mémoire. De la même manière, l'explication technique de l'opérateur de fusion en terme de manipulation de l'expression logique n'est pas abordée. Toutefois, si le lecteur est intéressé à en savoir plus, il peut trouver tous les détails dans [1, 2, 3]

2.3 Différentes approches de représentation de FMs

Il existe tout un éventail de solutions de gestion de FMs dont certaines sont décrites dans cette section (d'autres existent comme FaMa [29] ou encore S2T2 [52]). Les outils sélectionnés nous paraissent les plus représentatifs du marché et sont utilisés dans le cadre de ce mémoire. Certains sont graphiques, d'autres sont textuels et chacun a des avantages et des inconvénients, qu'il est parfois intéressant de combiner. Les concepts des FMs et les opérateurs de manipulations de ceux-ci sont définis dans la littérature (et repris en résumé

dans la Section 2.2), mais en pratique, l'implémentation donne lieu à des interprétations et à beaucoup de libertés.

2.3.1 Approches graphiques

Les approches dites « graphiques » sont des outils qui permettent de représenter à l'écran des FMs sous forme d'une hiérarchie de features. Elle consiste en un arbre dont les nœuds sont des features et les liens, des relations. Les différents types de relations sont illustrés par différentes représentation de liens. Pour la création ou la configuration d'un FM, des interactions graphiques sont envisagées, par exemple, le clic. Comme expliqué dans [37], le gros avantage avec ces approches est que c'est une façon intuitive d'aborder les FMs pour l'utilisateur. Il est donc facile pour un utilisateur non technicien de se familiariser avec l'outil. Cependant, construire un FM peut devenir une tâche lourde et qui prend beaucoup de temp car elle nécessite beaucoup de clics ou de glissés déposés. Il est donc très difficile de modéliser de grands FMs. De la même manière, naviguer dans un FM ou encore, le configurer devient une tâche difficile si celui-ci est d'une taille conséquente. De plus, ces approches se basent sur un programme particulier pour visualiser le modèle et manquent donc d'interopérabilité, ce qui implique un manque d'efficacité dans le travail collaboratif. Dans cette section, nous présentons deux gestionnaires de FM graphiques : FeatureIDE et S.P.L.O.T. Il en existe toutefois d'autres comme FAMA [29].

FeatureIDE

FeatureIDE [45, 59] consiste en un plugin eclipse crée en Java pour le développement de programmes orientés features. Cet outil est principalement graphique, mais a une syntaxe textuelle pour représenter des FMs. Il supporte toutes les phases de développement de SPLs : analyse du domaine, implémentation du domaine, analyse des exigences et la génération de programmes. Différentes techniques d'implémentation de SPL sont gérées comme la programmation orientée aspect [46] ou la programmation orientée feature [51]. Les *FMs basiques* sont pris en charge et, par conséquent, le *groupe Mutex*, les *attributs* et le *groupe à Cardinalité* ne sont pas représentables.

La création ou la modification d'un FM peut se faire de façon graphique, ce qui est très appréciable pour un utilisateur. Celui-ci peut, par exemple, ajouter ou modifier des features (voir la Figure 2.2), ajouter des relations, modifier les contraintes, etc. De la même manière, des facilités sont données pour configurer le FM (voir la Figure 2.3). Un arbre de features est présenté à l'utilisateur et il suffit de cliquer sur une feature pour la sélectionner, cliquer

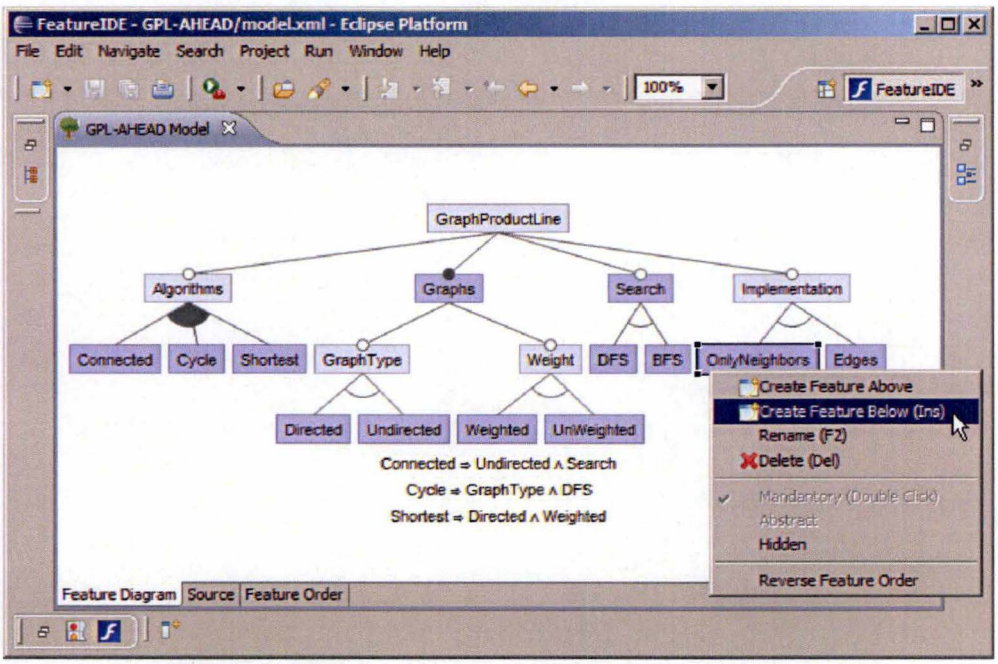

FIGURE 2.2 – Ajouter une feature à un FM avec FeatureIDE

une seconde fois pour la désélectionner et une troisième fois pour la retirer de la configuration. L'autopropagation sélectionne ou désélectionne les features sujettes aux différentes contraintes déclenchées.

S.P.L.O.T.

Comme décrit dans [47, 60], S.P.L.O.T. est un système de raisonnement et de configuration pour les SPLs. Celui-ci consiste en une interface web visant à mettre en pratique la recherche autour des SPLs. En effet, celui-ci permet aux gestionnaires de SPLs et aux chercheurs de créer, analyser et configurer des FMs en ligne (voir la Figure 2.4 pour la configuration) et de les partager avec la communauté. Cet outil est populaire, car il met à disposition des utilisateurs des catalogues de produits d'une taille relativement importante par rapport à ceux renseignés habituellement dans la littérature. Par exemple, le catalogue de VTTs contient 549 features et 376 produits.

Comme avec FeatureIDE, l'utilisateur peut concevoir ou configurer un FM en utilisant l'interface S.P.L.O.T.. La Figure 2.4 nous montre l'interface de configuration d'une ligne de produits d'ordinateurs portables. Il suffit à

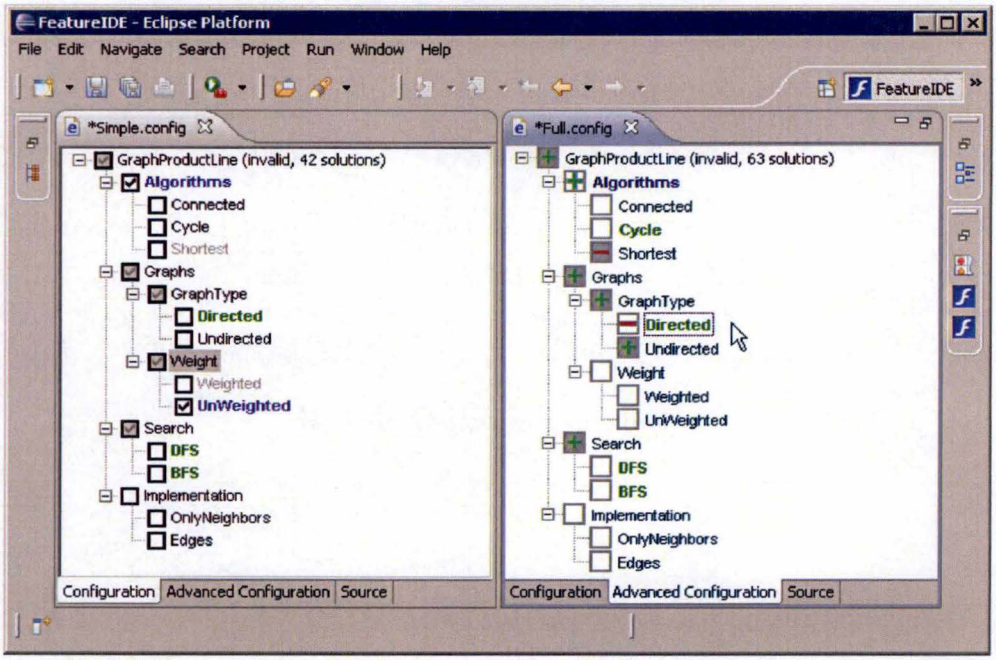

FIGURE 2.3 – Tâche de configuration d'un FM avec FeatureIDE

l'utilisateur de cliquer sur l'icône représentant un « V » vert ou un « X » rouge pour respectivement sélectionner ou désélectionner une feature. Les contraintes sont alors propagées automatiquement et la liste des configurations valides est réduite / étendue en temps réel.

S.P.L.O.T est implémenté en servlet Java et repose sur un moteur de raisonnement appelé S.P.L.A.R. Ces deux composantes ont été rendues open source en Novembre 2010 et ont une petite communauté de contributeurs, à la fois pour le dépôt public de SPLs ainsi que pour l'implémentation des fonctionnalités ou extensions. Un effort dans le développement de S.P.L.A.R. a été fourni pour l'optimisation des opérations de raisonnements en implémentant de façon performante, par exemple, la compilation des diagrammes de décisions binaires (ou BDDs) décrite dans [48].

2.3.2 Approches textuelles

Comme expliqué dans [37], les approches dites textuelles, sont très intéressantes dans la pratique. En effet, celles-ci ont de nombreux avantages. Elles ne nécessitent pas l'utilisation d'un outil particulier. Il est donc possible de modifier le FM avec n'importe quel éditeur de texte. Cela favorise la collaboration autour de la construction de FMs car, en plus de cela, des mécanismes de modularisation sont souvent associés avec ces langages. Il

FIGURE 2.4 – Tâche de configuration d'un FM avec S.P.L.O.T.

est donc possible d'éditer le FM à plusieurs et en même temps. De plus, il est, par exemple, possible d'employer des outils de versionnement comme Git [31], Subversion [57] ou CVS [20] pour maintenir les différentes versions de FMs. Les approches textuelles apportent aussi des concepts difficilement représentables graphiquement, comme les *attributs*, les contraintes composées, les commentaires, la spécialisation, les valeurs par défaut ou les types étendus. Malgré tous ces avantages, l'utilisateur devra se familiariser avec ces langages, ce qui représente un effort plus important que pour les approches graphiques. Cependant, selon [37], la courbe d'apprentissage est relativement courte.

TVL

TVL [8] est un langage textuel de modélisation de FMs avec une syntaxe ressemblante au C. Le but de celui-ci est de pouvoir monter en charge, par le fait d'être concis et en offrant un mécanisme de modularité et de réutilisation de structures. Le but principal est de rester compréhensif par le gestionnaire de la SPL en lui fournissant un grand ensemble de constructions (à forte sémantique) définies dans la littérature. Les *FMs à Cardinalités*, les *graphes orientés acycliques* (DAG ou Directed Acyclic Graph [44]), les *attributs* et diverses constructions supplémentaires (par exemple, les *énumérations*) sont gérées.

Les concepts et la syntaxe de ce langage ont été définis dans différents articles [16, 13, 17] et évalués dans [37]. Une implémentation a été proposée en Java et peut être téléchargée [8]. Celle-ci donne des facilités pour vérifier syntaxiquement le FM en entrée, calculer le nombre de features, effectuer un contrôle de satisfaisabilité, récupérer la liste des produits ou encore normaliser le modèle.

FAMILIAR

FAMILIAR [30, 5, 4] est un DSL [7] qui donne des facilités pour modéliser, manipuler et interagir textuellement avec des FMs. Il se cantonne aux *FMs de base*, formalisme peu expressif, pour pouvoir offrir des opérations riches de manipulation automatisée. Ce gestionnaire de FM a de particulier qu'il essaye de combiner les forces et les faiblesses de la majeure partie des autres gestionnaires de FM du marché en utilisant leur librairie.

FAMILIAR est particulièrement exposé dans cette section, car celui-ci sous-tend les opérations de raisonnement des différentes contributions de ce travail. En effet, il est le seul outil fonctionnel proposant l'opérateur de fusion (qui se révèle dans la pratique très utile [3]), condition sine qua non de notre outil d'extraction de la variabilité à partir de lignes de produits déjà existantes. De plus, le stage associé à ce mémoire s'est déroulé à l'école polytechnique de Nice Sophia-Antipolis, lieu où a été crée FAMILIAR. Vu les contacts étroits avec l'équipe de développement, il est tombé sous le sens de l'utiliser.

Il est possible d'interagir avec FAMILIAR en ligne de commande classique ou avec le plugin eclipse qui comporte, en plus, une visualisation graphique. Ce plugin est le résultat de l'intégration de l'environnement FeatureIDE. Toutes les modifications des FMs via la partie graphique sont synchronisées avec les variables de l'environnement FAMILIAR ainsi que toutes les opérations FAMILIAR le sont avec l'éditeur graphique. D'autres types de visualisations sont implémentés de manière expérimentales, comme S.P.L.O.T. ou encore Zest [64].

FAMILIAR mise sur l'interopérabilité, un certain nombre de formats d'entrée ou de sorties sont gérés. Les fichiers actuellement pris en charge sont ceux de S.P.L.O.T. (.xml), FeatureIDE (.m), TVL (.tvl [8]), Triskell (.featuredia-

7. Les DSLs (Domain Specific Langage) sont des langages adaptés à un domaine applicatif spécifique. Ils offrent un gain d'expressivité et une facilité d'utilisation comparé aux langages généralistes. Le pourquoi et le comment de l'utilisation de ce type de langage est discuté dans [49].

8. Un sous-ensemble est géré. Ce module d'import fait partie d'une des contributions

gram) ainsi que bien sûr, FAMILIAR (.xmi et .fml). Il est donc possible de créer un FM en TVL, avec toutes les facilités de modélisation que ce langage fournit, pour ensuite le manipuler avec FAMILIAR.

FAMILIAR a été développé en Java et utilise Xtext [63], un framework pour le développement de DSLs. Le moteur de raisonnement se base sur des librairies de solveur SAT [53] et de BDD [39]. L'utilisation de deux librairies plutôt qu'une se fait par souci d'optimisation. En effet, le décompte des configurations se fait via les BDDs tandis que les autres opérations se font par le solveur SAT. De la même manière, il est possible de paramétrer FAMILIAR pour qu'il utilise S.P.L.A.R. pour certaines opérations de raisonnement.

Ce DSL, avec une courbe d'apprentissage relativement courte, donne un environnement complet et des fonctionnalités de manipulations des FMs. La combinaison d'opérateurs sous forme de scripts permet la réutilisation et la collaboration des utilisateurs autour de la SPL. Des facilités sont données à l'utilisateur pour interagir de manière programmative avec FAMILIAR. Par exemple, FAMILIAR a été utilisé pour maintenir en mémoire l'état (la configuration) ainsi que la variabilité autour de la SPL qu'est la vidéo surveillance [6]. L'utilisation des opérations de configuration et l'autopropagation permet de faire évoluer la configuration des caméras de sécurité. Des tests synthétiques, des cas d'utilisations ainsi que la contribution principale de ce mémoire, évaluée au Chapitre 8, ont montré la montée en charge de FAMILIAR pour des FMs de taille conséquentes.

FAMILIAR donne des facilités d'interopérabilité, un environnement complet d'opérateurs de manipulation et de raisonnement à propos de FMs. Cependant, ce langage, en tant que langage de modélisation manuel de FMs, est peu expressif, car il ne représente que les *FMs basiques*. En effet, les constructions gérées ne comportent pas, par exemple, des *FMs à Cardinalité*, des sucres syntaxiques comme des *énumérations*, deux features de même nom, les *DAGs*, ou encore les *attributs*. Il peut donc être intéressant, vu la modularité de FAMILIAR, d'utiliser un autre langage de modélisation comme TVL en entrée et de l'importer.

de ce mémoire et est décrit à la Section 5.2

Chapitre 3

Le cas WikiMatrix

Dans ce chapitre, nous donnons un cas d'utilisation qui servira de base aux exemples du reste de ce mémoire. Les données utilisées proviennent de Wikimatrix [61], une organisation qui fournit un accès à des informations détaillées à propos de multiples moteurs de wiki. La liste des fonctionnalités supportées/offertes par les moteurs de wiki est documentée en utilisant le formalisme XML [62], ou plus généralement, une spécification semi-structurée [1].

Pour notre exemple, nous avons choisi de sélectionner seulement un extrait des données fournies par WikiMatrix. En effet, la base de donnée fournie est assez large : elle regroupe la comparaison de plus de 130 wikis sur 140 critères, ce qui donne plus de 18000 valeurs de comparaison. Pour donner un exemple intéressant et raisonnable en taille, nous avons filtré les données et nous avons retenu 8 moteurs de wiki et 11 critères de comparaisons que nous trouvions pertinent. Nous les avons classés en 3 points de vues, appelés : *General*, *Functionnalities* et *Requirements*. La Figure 3.1 regroupe les caractéristiques dites générales (*General*) comme le langage (Language) de programmation ou le moteur de stockage (Storage), c'est-à-dire si les données sont stockées sous la forme de fichiers, de base de données, etc. La Figure 3.2 regroupe les fonctionnalités (*Functionnalities*) du moteur de wiki. La Figure 3.3 regroupe les caractéristiques dites prérequises (*Requirements*) à l'installation.

Pour représenter et manipuler plus facilement les données, nous avons écrit un convertisseur entre les fichiers XML donnés par WikiMatrix et des

1. Nous qualifions les données *semi-structurée* de par le fait que ce sont des données structurées, mais qui ne sont pas conformes avec la structure formelle de table et de modèle de données associées aux bases de données relationnelles, mais elles contiennent toutefois des marqueurs pour séparer les éléments sémantiques [14].

structures tabulaires à 2 dimensions. Des fichiers CSV sont utilisés pour contenir ces données, car ils sont particulièrement indiqués pour représenter des structures tabulaires. Le format de fichier CSV est un type de données semi-structurées. Un fichier CSV consiste en un fichier textuel (en opposition aux fichiers binaires) dont les valeurs sont délimitées par un séparateur comme, par exemple, des virgules. Vu son manque de définition précise, son utilisation est plutôt libre. Toutefois, nous avons pris comme point de départ [56] pour définir nos propres règles constituant un fichier CSV. Les Figures 3.1, 3.2 et 3.3 fournissent une représentation visuelle de ces 3 fichiers ou points de vues. Chacune des lignes documente un produit (dans ce cas, un moteur de wiki) ainsi que la liste des fonctionnalités supportées. Par exemple, le wiki *Confluence* a une licence Commerciale qui coute 10 Dollar et supporte les flux RSS. Nous pouvons aussi remarquer que *DokuWiki*, *PmWiki*, *DrupalWiki* et *MediaWiki* sont développés en PHP.

Identifier	License	Language	Storage	LicenseFee
PBwiki	Nolimit	No	No	Yes
MoinMoin	GPL	Python	Files	No
DokuWiki	GPL2	PHP	Files	No
PmWiki	GPL2	PHP	Files	No
DrupalWiki	GPL2	PHP	Database	Differentlicences
TWiki	GPL	Perl	FilesRCS	Community
MediaWiki	GPL	PHP	Database	No
Confluence	Commercial	Java	Database	US10

FIGURE 3.1 – Point de vue *General*

Identifier	RSS	Unicode	Flash	ImageEditing
PBwiki	Yes	No	Indentedblock	Yes
MoinMoin	Yes	Yes	Yes	No
DokuWiki	Yes	Yes	Yes	Yes
PmWiki	Yes	Yes	Yes	Yes
DrupalWiki	Yes	Yes	Yes	Yes
TWiki	Yes	Yes	Yes	Yes
MediaWiki	Yes	Yes	Yes	No
Confluence	Yes	Yes	Yes	Yes

FIGURE 3.2 – Point de vue *Functionnalities*

En se basant sur la spécification du catalogue de produits, un gestionnaire de SPL peut considérer que les 8 moteurs de wiki forment une SPL. Nous

Identifier	OperatingSystem	RootAccess	Other
PBwiki	No	Yes	N_A
MoinMoin	No	No	No
DokuWiki	No	No	No
PmWiki	No	No	No
DrupalWiki	Yes	Yes	No
TWiki	No	No	Yes
MediaWiki	N_A	Yes	No
Confluence	No	No	Java5

FIGURE 3.3 – Point de vue *Requirements*

prenons comme scénario qu'il veut produire un modèle qui représente les éléments communs et la variabilité des 8 produits. La relation entre une SPL et un FM peut être résumée comme suit : une SPL est un ensemble de produits où chacun d'eux correspond à une configuration valide d'un FM. Un tel modèle peut être utilisé comme point de départ au développement de la SPL, pour par exemple, raisonner à propos des variantes ou identifier les frontières de celle-ci. La génération de FM à partir de catalogue de produits existants est discutée dans le Chapitre 4. Dans le reste de ce mémoire, nous allons considérer qu'une configuration est définie par un ensemble de features sélectionnées. Par exemple : { General, License, GPL, Language, Python, Storage, Files } est une configuration valide du FM représenté à la Figure 3.4 et correspond au moteur de wiki *MoinMoin*. Le FM extrait contient tous les produits du point de vue *General* de la Figure 3.1. La Figure 3.4 comprend uniquement un extrait de contraintes générées par souci de lisibilité.

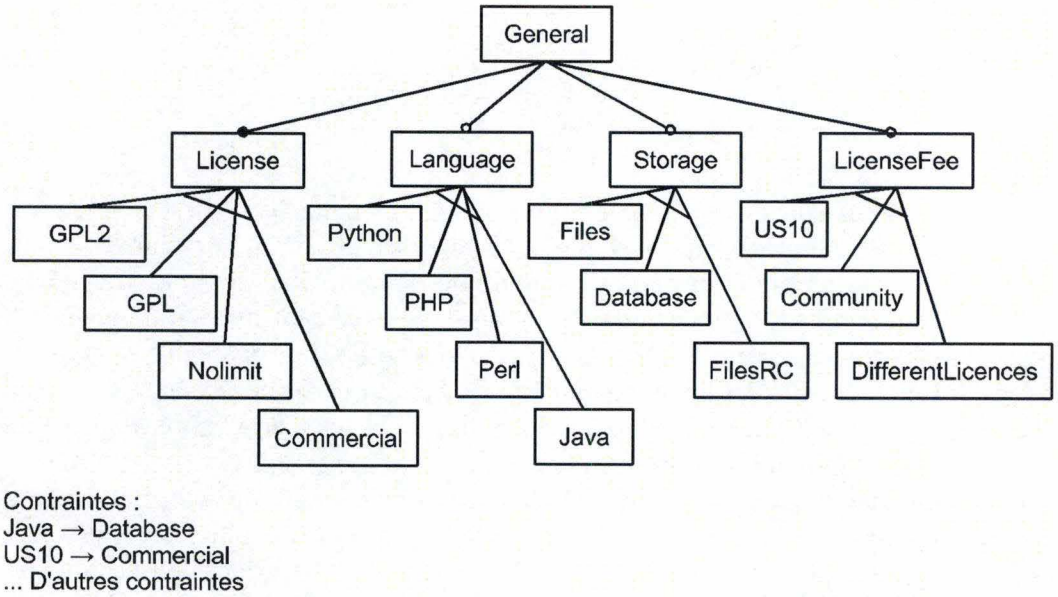

FIGURE 3.4 – FM extrait de la spécification des produits de la figure 3.1

Chapitre 4

Extraction de Feature Models à partir de catalogues de produits

Pour concevoir et implémenter une SPL, une analyse du domaine est habituellement accomplie. Une portion significative de ce domaine de connaissance est parfois disponible sous forme digitale. Par exemple, des données tabulaires (des feuilles de calcul) sont très largement utilisées pour de nombreuses tâches professionnelles et sont susceptibles de contenir un large éventail de savoirs implicites comme la spécification des produits ou *des catalogues de produits.* La tâche de construire des FMs peut être très ardue pour les parties prenantes, et ce, particulièrement s'ils doivent être déduits à partir de documents peu structurés. En effet, une large quantité d'information du domaine du SPL doit être collectée, modelée, et plus tard interprétée en terme de features et de variabilité [40, 41]. De plus, les opérations de raisonnements sur les FMs produits comme celle d'identifier les *features principales* ainsi que les *features qui varient* est particulièrement lente et constitue une lourde tâche. Une difficulté supplémentaire est que les combinaisons de features ne sont pas forcément toutes valides. En effet, elles ne correspondent pas forcément à un produit de par les contraintes qui restreignent le domaine. Cette tâche est extrêmement difficile quand elle est effectuée manuellement, c'est pourquoi l'extraction aidée par un outil va significativement aider les intervenants en rendant le processus d'extraction moins sujet aux erreurs, plus robuste et considérablement plus rapide.

Vu la qualité intrinsèque et le support outil existant des FMs, nous les utilisons dans notre approche. Nous motivons leur emploi à la Section 4.1. À la Section 4.2, nous identifions les défis rencontrés pour la construction d'un « bon » FM qui représente efficacement le catalogue de produit donné en entrée. Pour transformer un ensemble de fichiers contenant des données

semi-structurées en un FM, nous avons développé un processus décrit dans la Section 4.3.3. Celui-ci utilise un modèle de conversion pour normaliser les données en entrée et le traduit en un script exécutable. Ce script, une fois chargé dans un gestionnaire de FM, produit le FM final qui est une représentation compacte de la SPL.

4.1 D'un catalogue de produits à une représentation compacte

Le but de notre approche est décrit par la Figure 4.1. Afin d'obtenir une représentation compacte de toutes les combinaisons valides de features décrites dans un catalogue de produits (exemple : comme celui de la Figure 3.1 ou des fichiers XML,etc.), nous passons par un modèle de conversion pour normaliser les données en entrée. Nous donnons des facilités à l'utilisateur pour le construire et l'adapter en fonction de ses besoins. Une fois construit, ce modèle peut être converti de deux façons. L'une est dite logique, car elle consiste en une formule booléenne avec comme variables les différentes features. L'autre est une hiérarchie de features et représente l'information de variabilité.

D'une part, nous soutenons qu'une représentation logique est suffisante. Par exemple, quand un gestionnaire de SPL veut déterminer si une certaine combinaison de features, disons f_1 et f_2, conduit à au moins un produit, un solveur SAT ou une librairie de BDD peuvent être utilisés pour raisonner dans l'espace logique. Des vérifications de consistance peuvent donc être opérées sur $\phi_{catalog} \wedge f_1 \wedge f_2$, où $\phi_{catalog}$ est la formule booléenne qui représente les combinaisons valides du catalogue de features. De plus, le domaine de validité peut être calculé pour inférer certains choix de variabilité.

D'autre part, nous soutenons qu'en plus de la représentation logique, une hiérarchie de features et l'information de variabilité associée (exemple : un FM [1]) est nécessaire. Cette hiérarchie est particulièrement utile lorsque le raisonnement sur la ligne de produit est accompagné d'une activité complexe de gestion. Deelstra et al. [27] rapportent que la *configuration* de produit est une activité coûteuse et qui prend beaucoup de temps. Leurs résultats montrent notamment que l'absence de regroupement approprié des points de variation est un grave obstacle à une configuration de produit efficace. De plus, une hiérarchie propre serait utile pour comprendre des relations complexes entre les features, par exemple, durant la configuration de produits parmi des ensembles de produits en compétition. De manière similaire,

1. Comme déclaré dans [24], l'essence d'un FM est l'incarnation d'une hiérarchie et la description de la variabilité, plutôt que son rendus. Par hiérarchie, nous ne voulons ni désigner des arbres FODA, ni des vues d'explorations.

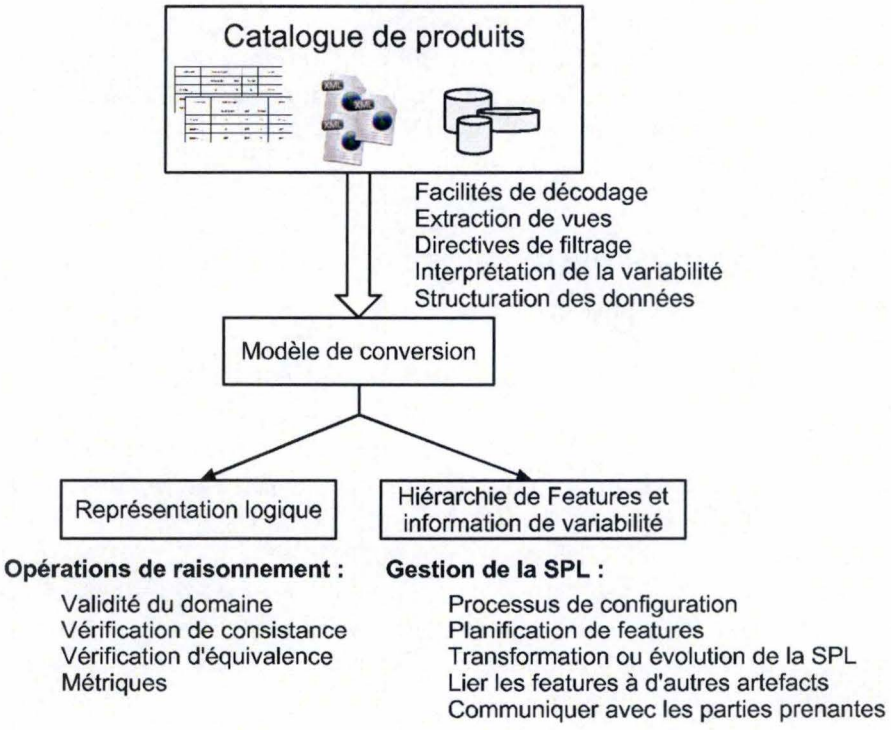

FIGURE 4.1 – Scénario d'extraction

l'*évolution* (refactoring, généralisation et spécialisation) d'un SPL existant est une activité difficile [38, 58] et le FM extrait, via notre approche, peut servir de point de départ pour une évolution future par le gestionnaire de SPL. Durant le développement d'une SPL, le FM extrait peut être associé à d'autres artefacts comme du code ou des modèles [23, 33]. Un autre argument est que quand les parties prenantes (d'une division d'une entreprise ou des fournisseurs externes) *communiquent* à propos de la planification de features et des réutilisations opportunistes, ils peuvent avoir une documentation explicite de la variabilité, c'est-à-dire un FM. La littérature sur l'analyse automatisée de FMs donne un large ensemble d'opérations, de techniques, d'outils et de résultats empiriques. De tels mécanismes automatisés peuvent être utilisés par les gestionnaires de SPL pour extraire de l'information utile à partir de FMs [11] (par exemple : calculer des métriques).

En vue de tous ces arguments, nous avons décidé de baser notre approche sur les FMs plutôt que sur une représentation logique. Il est toutefois possible de récupérer cette dernière, car elle sous-tend la construction des FMs.

4.2 Les défis de l'extraction de FMs

La création manuelle d'un FM qui contient la variabilité associée à un ensemble de produits est une tâche fastidieuse et sujette à erreurs. La hiérarchie de features (ou arbre) n'est pas le seul élément du FM à considérer. En effet, déterminer si une feature est obligatoire, optionnelle ou dans un groupe alternatif est loin d'être une tâche immédiate. De plus, les contraintes complexes liants certaines features entre elles sont difficiles à déduire par un humain. Par exemple, dans la Figure 3.4, la contrainte « Java implique Database » est *déduite* de la spécification de produits : il n'y a pas de moteurs de wiki qui est basé sur java et qui supporte une stratégie différente de stockage que les bases de données. Cette tâche devient encore plus difficile lorsque le FM est composé de multiples perspectives ou *préoccupations*. Par exemple, la spécification de moteurs de wiki comprend les perspectives General (Figure 3.1), Functionnalities (Figure 3.2), Requirements (Figure 3.3), etc. Dans ce cas, l'utilisateur doit considérer des sources différentes et potentiellement hétérogènes.

L'idée d'extraction d'un FM à partir d'un catalogue de produit rencontre certains défis :

Défi 1 : Automatisation de l'extraction. Comme énoncé ci-dessus, nous voulons réduire l'effort de l'utilisateur et le temps nécessaire à la construction d'un FM à partir d'un catalogue de produit. Le problème clé est l'automatisation de cette activité pour accélérer le développement.

Défi 2 : Insérer l'utilisateur dans le processus d'extraction. Bien que la procédure doit être complètement automatisée, l'utilisateur devrait être capable de jouer un rôle dans l'extraction. Comme dans l'ingénierie des SPLs, un problème clé est de donner des facilités pour le filtrage. Par exemple, il peut être utile de ne pas considérer certains produits, points de vues ou features dans le FM résultant de l'extraction. De plus, la manière dont les données sont interprétées en terme de variabilité doit être paramétrable, si le besoin est, pour mieux correspondre à l'intention de l'utilisateur.

Défi 3 : Qualité des FMs extraits. Une propriété requise du FM extrait est qu'il représente avec précision les combinaisons valides de features supportées par les produits intégrés à la SPL. Cependant, ce n'est pas le seul critère vu que différents FMs peuvent représenter le même ensemble de configurations. La conséquence est que générer une structure porteuse de sens à la hiérarchie de features (incluant des relations parent-enfants et des *groupes Ou / Alternatifs*) pourrait aider l'utilisateur et éviter le refactoring manuel [38] du FM.

Défi 4 : Montée en charge de l'extraction. L'extraction devrait être capable de gérer un grand ensemble de données pour être utilisable dans

l'industrie. Des limites théoriques ou pratiques devraient être identifiées en terme de nombre de ressources, de produits, de features, etc.

Pour le défi 2, dans la Section 4.3.2, nous allons montrer comment le gestionnaire de SPL peut paramétriser le convertisseur. Ensuite, nous allons décrire le processus d'extraction et montrer comment celui-ci peut être complètement automatisé (voir la Section 4.3.3) pour produire un FM satisfaisant. Nous allons aussi proposer une solution qui peut monter en charge et son évaluation en considérant les défis 3 et 4 mentionnés ci-dessus (voir la Section 8).

4.3 Processus d'extraction

La Figure 4.2 nous montre les différentes étapes de l'extraction que nous allons décrire dans cette section. Premièrement, l'utilisateur doit transformer ses fichiers en un modèle de conversion ①, défini à la Section 4.3.1. Celui-ci représente les données, les structure autour de concepts et donne une sémantique aux constructions. Ensuite, ② l'utilisateur écrit un ensemble de directives dans un langage dédié (ou DSL) qui filtre le modèle utilisé. Les instructions d'organisation des données et de la variabilité avec le DSL sont présentées dans la Section 4.3.2. Avec les facilités données par le DSL, l'utilisateur peut adapter le modèle à ses besoins et lui donner la structure voulue. Une fois construit, ③ le modèle de conversion est donné au convertisseur qui ④ se charge de traduire ce modèle en un script FAMILIAR en 3 étapes. Elles sont décrites à la Section 4.3.3. Ce script, ou fichier FML, sera ⑤ exécuté par l'implémentation FAMILIAR pour produire le FM demandé.

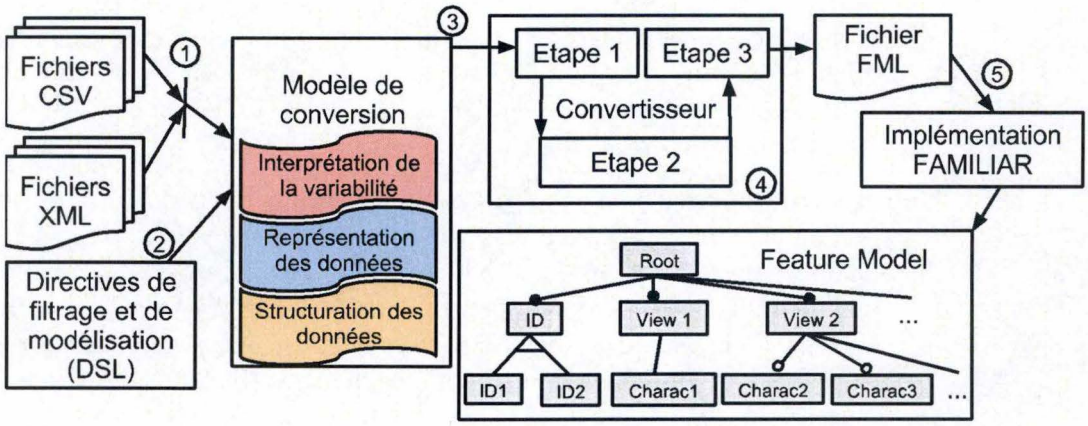

FIGURE 4.2 – Processus d'extraction complet

4.3.1 Modèle de conversion

Le but du processus d'extraction est de transformer des données semi-structurées en un FM unique qui représente et organise la variabilité. Dans leur forme syntaxique pure, les données semi-structurées ne sont pas suffisamment riches pour avoir une traduction directe en un FM. La clé de la solution consiste en l'*interprétation* des données en terme de variabilité. Les points ① et ② de la Figure 4.2 représentent les données et directives en entrée du convertisseur. Nous proposons un modèle de conversion générique en vue de les normaliser et de faciliter leur interprétation. En particulier, nous réifions les concepts de modélisation de feature pour la conversion et nous définissons des règles de transformation (exemple : déterminer dans quelles conditions une valeur correspond à une feature obligatoire). Pour un formalisme en entrée, le gestionnaire de la ligne de produits écrit un convertisseur entre son formalisme et le modèle de conversion (celui pour CSV est fourni dans notre implémentation). En se basant sur le modèle de conversion, nous fournissons un langage dédié pour spécifier des directives (exemple : de filtrage) de haut niveau et pour paramétriser la transformation des données.

De par la simplicité du format de fichier CSV, celui-ci est fort utilisé, car il permet pour ainsi dire de traduire n'importe quel formalisme. On peut donc intuitivement et facilement traduire un fichier de tableur, un fichier XML ou une table d'une base de données relationnelle en un ou des fichiers CSV. Ces trois formats étant très souvent utilisés dans la pratique pour maintenir un catalogue de ligne de produits, l'utilisation de CSV et plus particulièrement des tableaux à deux dimensions prend tout son sens.

Un FM n'étant à la base qu'un ensemble de features avec des relations et des contraintes, il nous faut définir des concepts, c'est-à-dire, un modèle permettant de transcrire le catalogue de produits en un FM significatif et bien structuré. C'est pourquoi nous allons vous présenter dans cette section un modèle de conversion entre un ensemble de tableaux à deux dimensions comme, par exemple, des fichiers CSV, et un FM. Ce modèle de conversion structure le FM autour de concepts. De plus, il donne à l'utilisateur des facilités pour manipuler et adapter les données en entrée (souvent brutes) à ses besoins.

Le modèle de conversion est décrit par le diagramme de classe de la Figure 4.3. Comme indiqué par le point ③ de la Figure 4.2, ce modèle peut être divisé en trois parties : (1) les éléments en *bleu* correspondent à un tableau à deux dimensions avec de la sémantique ajoutée. Il normalise les données en entrée ; (2) les éléments en *rouge* correspondent aux différentes valeurs contenues dans les tableaux, leur interprétation et leur conversion en terme de features et contraintes ; (3) les éléments en *orange* correspondent

aux structures et relations permettant une cohérence entre les éléments du tableau, les valeurs et les produits dans le FM à obtenir. Il est nécessaire d'indiquer que les relations Is-a du diagramme sont des partitions.

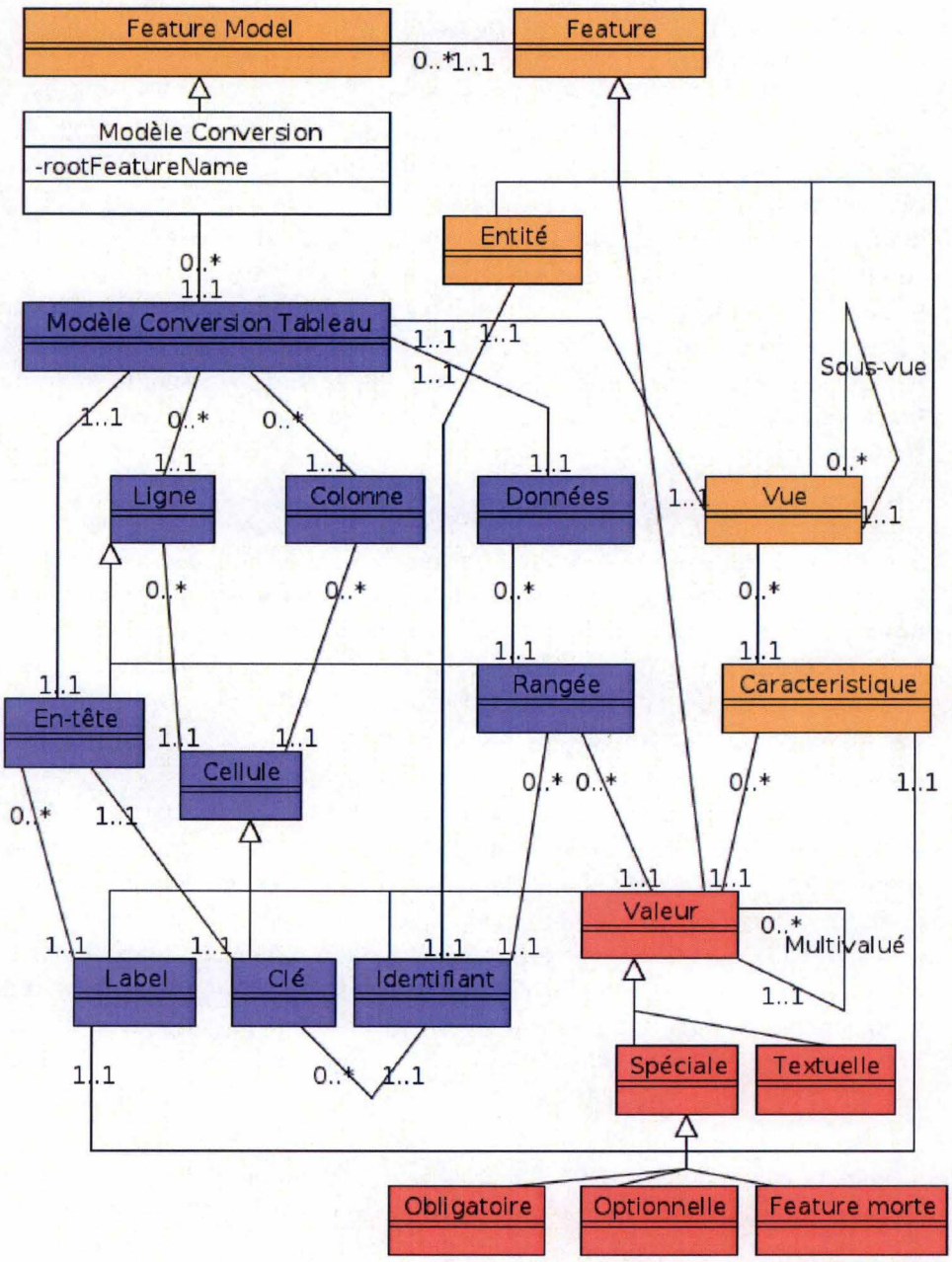

FIGURE 4.3 – Diagramme de classe du modèle de conversion

Modélisation des données

La partie bleue du modèle de conversion consiste en un tableau à deux dimensions. Celui-ci contient des *Lignes* (horizontalement) et des *Colonnes* (verticalement). Une *Cellule* est le contenu placé à l'intersection d'une ligne et une colonne. La première ligne du tableau est l'*En-tête*. Les cellules de l'en-tête sont nommées *Labels* [34]. Chaque label doit être unique dans l'entièreté du modèle de conversion. Une *Rangée* est une ligne autre que l'en-tête. L'ensemble de toutes les rangées constitue les *Données*. Un des labels est appelé *Clé*. La clé sert à indiquer que le contenu des cellules de cette colonne identifient (de manière unique) les rangées associées dans la table. Elles sont nommées *Identifiants*. Une *Valeur* est le contenu d'une cellule des données autre qu'identifiante.

Dans la Figure 3.1, la première ligne consiste en l'en-tête, la clé est la cellule Identifier et les labels sont entre autres License et Language. Les identifiants sont les cellules associées à la clé. Les cellules *Confluence* et *PBwiki* identifient donc leur rangée et sont associées à un produit de la SPL. Pour l'identifiant *PBwiki* et le label License, la valeur associée est NoLimit. C'est-à-dire, avec une interprétation, que le moteur de wiki *PBwiki* a une licence non limitée (non-propriétaire).

Un utilisateur sait facilement transformer un formalisme quelconque en ce modèle de conversion. Avec le format de fichier CSV, c'est pour ainsi dire immédiat. Convertir un fichier CSV vers ce modèle pour obtenir un FM est utile, mais non suffisant pour des cas d'étude réalistes. En effet, comme dans les bases de données relationnelles, utiliser uniquement une seule table n'a pas de sens. Il est donc nécessaire de combiner la variabilité d'un ensemble de tableaux en un seul FM. Pour ce faire, nous définissons un ensemble de concepts servant de points de pivot pour la combinaison des données en entrée : *Vue*, *Entité* et *Caractéristique*. Ces concepts font la traduction entre la structure en tableau (les concepts de la partie bleue du diagramme) et leur traduction en terme de feature et contrainte dans le FM final. Ils sont définis dans la partie orange du diagramme de classe de la Figure 4.3.

Dans la Figure 3.1, nous pouvons considérer les identifiants comme des entités. Ils sont uniques (dans l'entièreté du modèle de conversion) et représentent un concept ou un objet, réel ou non. Ce sont les identifiants des différents produits de la SPL. Le FM final contient l'union de chacun des identifiants de l'ensemble des tableaux. Une vue, correspondant à un tableau (défini par la section bleue du diagramme de classe, consiste en un fichier, un point de vue de l'utilisateur), a un nom qui illustre les données contenues dans celui-ci. Elle contient des caractéristiques (ou labels) qui sont associées à une colonne et définissent les entités de la vue. L'association d'une carac-

téristique et une valeur caractérisent l'entité correspondante. Les différentes vues donnent des niveaux de granularité aux données contenues dans le FM et structure hiérarchiquement les différentes caractéristiques. Il en découle donc le principe de sous-vues, c'est-à-dire la possibilité de créer plusieurs niveaux de vues. Cette extension à notre approche sera développée à la Section 5.1.1.

Nous pouvons considérer comme 3 vues distinctes les tableaux des Figures 3.1, 3.3 et 3.2 faisant partie du modèle de conversion. Les entités *PBwiki*, *MoinMoin*, *DokuWiki* sont caractérisées dans les 3 vues. Par exemple, la caractéristique RSS avec la valeur associée Yes pour l'entité *Confluence* signifie que « *Confluence* gère les flux RSS ».

Interprétation des données

Du point de vue des SPLs, nous avons remarqué que certaines valeurs du tableau ont plus de sémantique que de simples valeurs textuelles. Le gestionnaire de SPL voudrait par exemple rendre une caractéristique optionnelle pour une entité et une vue donnée. Pour ce faire, celui-ci va intuitivement donner Opt (ou N/A, Optional... ou un autre texte donné) comme valeur. Nous avons identifié quatre différentes constructions. Elles sont définies dans la partie rouge du diagramme de classe donné à la Figure 4.3.

- Relation *Obligatoire* pour les valeurs suivantes données par l'utilisateur : « Yes », « 1 »,...
- Relation *Optionnelle* pour les valeurs suivantes : « N/A », « Opt »,...
- Une *feature morte* pour la valeur : « No ». Une feature morte est sémantiquement différente d'une absence de feature. En effet, une feature qui n'est pas présente n'a pas le même sens qu'une feature qui est toujours désélectionnée.
- Nous créons une sous feature contenant toute autre valeur textuelle qui n'a pas de sémantique particulière. Celle-ci sera marquée *Obligatoire*.
- Dans certaines circonstances, une valeur peut être multivaluée. En terme de variabilité, elle peut être interprétée de multiples manières. Par exemple, pour *Windows;Linux;MacOS*, les features Windows, Linux et MacOS peuvent être considérées comme étant soit mutuellement exclusives (dans un *groupe Alternatif*), soit *Optionnelles*, soit *Obligatoires*, etc. C'est pourquoi nous fournissons à l'utilisateur la possibilité, via le DSL, de définir une sémantique qui correspond à ses intentions initiales. Le type multivalué des variables est considéré comme une extension du processus d'extraction. Elle est décrite en détail à la Section 5.1.3.

La construction détaillée de ces relations est définie à la Section 4.3.3.

4.3.2 Organisation des données et de la variabilité

Notre approche consiste en la transformation des données et donc du modèle de conversion en un FM générique. Cependant, il est possible que ce modèle ne corresponde pas exactement aux besoins de l'utilisateur. C'est pourquoi notre approche donne des facilités à celui-ci pour gérer à sa manière le processus d'extraction. Nous lui fournissons un langage dédié [2] pour qu'il puisse paramétriser programmativement le processus d'extraction.

Grâce aux directives du Listing 4.1 et les données brutes sous format CSV de Wikimatrix, nous obtenons un modèle de conversion contenant 4 vues : *General*, *Deployment*, *Hosting* et *datastorage*.

```
1  // Import des données
2  import: "general.csv" as general
3      // Import de "general.csv" dans un tableau du modèle de
           conversion nommé general
4      "datastorage.csv" as datastorage
5      "hosting.csv" as hosting
6      "security.csv" as security
7
8  name: "WikiCompare"
9  // Nom de la feature racine du FM final
10
11 // Structuration des vues
12 structure:
13     security moveIn hosting
14     hosting below general
15     deployment abstract datastorage
16
17 // Directives à portée globale, sur toutes les vues
18 default:
19     parsing:
20         key: "Identifier" // Label sous lequel se trouve les
               identifiants
21         // Interprétation de la variabilité, détermination des
               valeurs textuelles correspondant aux constructions à
               obtenir
22         variability:
23             optional: "Optional" "Opt" "Partial"
24             mandatory: "Yes" "true" "1"
25             notpresent: "No" "false" "0" // Connu aussi comme
                   feature morte
26         separator: "," // Séparateur des valeurs CSV
27         multivalued_separator: ";" // Séparateur des valeurs
               multivaluées
28     // Filtrage des entités pour garder uniquement celles
           définies ici
29     only_products: "Moinmoin" "DokuWiki" "PmWiki"
```

2. Le langage est utilisé pour paramétriser l'entrée de la ligne de commande

```
30
31 // Directives à portée locale, sur la vue general
32 view: general
33    rootname: "General" // Nom de la vue
34    // Surcharge à portée locale de l'interprétation de la
          variabilité
35    parsing:
36       variability:
37          optional: "Plugin" "Optional" "Opt" "Partial"
38    // Elimines les caractéristiques suivantes de la vue
39    except_features: "Version" "Author"
40    // Réécriture d'une valeur textuelle en une autre
41    rewriting: "License Cost/ Fee" => "LicenseCost"
42    // Interprétation des valeurs multivaluées
43    multivalues:
44       "Other Limits" => Alternatives
45       "Intended Audience" => OR-Alternatives
46
47 view: security
48    rootname: "Security"
49       only_features: "Mail Encryption" "Blacklist"
50
51 view: datastorage
52    // Réécriture de valeurs. Remplace dans la tableau les
          valeurs "MySQL" ou "PostgreSQL" ou "SQLite" par la valeur
          "SQL"
53    structure: "SQL" replace "MySQL" or "PostgreSQL" or "SQLite"
54    multivalues: "Storage Quota" => Alternatives
```

Listing 4.1 – Exemple de création d'un modèle de conversion avec les directives du DSL fourni

Comme vous pouvez le voir, les directives sont structurées en plusieurs parties :

- Import : Lignes 2 à 6 : Permet de spécifier les différents fichiers à importer et les associer à une variable.
- Structure : Lignes 12 à 15 : Permet de donner une structure particulière au FM final. Permet, par exemple, de déplacer des données entre des vues, de définir des sous-vues, etc.
- Default : Lignes 18 à 29 : Spécifie un ensemble de filtres appliqués à l'ensemble des vues. Permet, par exemple, de restreindre les entités à un sous-ensemble donné.
- Les différentes vues : Lignes 32 à 54 : Ensemble de définitions de filtres pour des vues spécifiques. Les filtres définis dans ces différentes parties auront un impact local, c'est-à-dire, un impact dans la vue associée. Par exemple, les lignes 47 à 49 concernent la vue *security*.

Le DSL donne une interaction conviviale avec les fonctions du modèle de conversion de la Figure 4.3. Il donne certaines facilités pour :

Décoder l'entrée. (1) Un gestionnaire de SPL peut importer différentes sources de données avec la directive import (lignes 2 à 5 du Listing 4.1). Elles correspondent aux différentes perspectives, intérêts et (sous-)domaines ; ce que nous appelons vues. Par conséquent, chaque source de données est associée à une vue ; (2) les produits/entités sont identifiés par une clé et le praticien peut définir, par exemple, à quelle colonne du tableau la clé correspond. Pour ce faire, faut utiliser la directive key ("key")* avec key, les labels identifiants (ligne 20) ; (3) l'interprétation des valeurs en terme de variabilité et de construction traduite peut être spécifiée. Les 4 types de constructions prises en charge sont : optional, mandatory et notpresent[3](lignes 23 à 25) ; (4) certaines instructions syntaxiques de décodage des données en entrée peuvent être spécifiées. Par exemple, dans le format CSV, il est nécessaire de spécifier le séparateur des différentes valeurs avec separator (ligne 26) ainsi que le séparateur des champs multivalués avec multivalued_separator (ligne 27).

Filtrer les données. Un gestionnaire de SPL pourrait vouloir limiter la portée des données ou les filtrer de multiples manières et pour de multiples raisons. Par exemple, toutes les vues ne sont pas à prendre en considération pour décrire la SPL. En effet, il se peut que le (sous-)domaine soit non pertinent pour l'utilisateur ou bien que la granularité de l'information soit trop importante. L'entièreté des produits ne doit pas non plus forcément être intégrée dans le FM. Certains produits pourraient être trop pauvres en terme de features supportées et pas assez compétitif. De plus, l'entièreté des caractéristiques ou features ne doit pas forcément être considérée. Par exemple : le numéro de version du produit n'est pas une information pertinente en terme de prise de décision à partir d'un FM. Les directives except_products et only_products (ligne 29) du DSL définissent les produits a ou a ne pas considérer pour l'ensemble des vues importées (exemple : nous considérons uniquement les entités *Moinmoin*, *DokuWiki* et *PmWiki*). En outre, les directives except_features (ligne 39) et only_features (ligne 49) peuvent être spécifiées pour chacune des vues, localement. Par exemple, à la ligne 49, nous utilisons la directive only_features pour spécifier que nous sommes uniquement intéressés aux caractéristiques Mail Encryption et Blacklist dans la vue *Security*.

Transformer les données. Ces directives consistent en des fonctionnalités de renommage / réécriture ou des opérations plus complexes de remplacement[4]. Par exemple, « *"SQL" replace "MySQL" or "Post-*

3. Dans ce mémoire, cette construction est aussi appelée feature morte.

4. Il faut remarquer que, plus généralement, les données peuvent être prétraitées en utilisant les processus d'Extraction, Transformation et chargement (Loading) (ETL – voir `http://en.wikipedia.org/wiki/ETL`). Nous avons décidé d'intégrer seulement un extrait

greSQL" or "SQLite" » à la ligne 53, signifie que la feature SQL sera présente dans le FM résultant si la spécification du produit contient soit la feature MySQL, PostgreSQL ou SQLite.

Spécifier la structure. Les vues sont probablement liées les unes aux autres (pour décrire un sous-domaine) et les informations structurantes ne sont pas implicites avec la structure tabulaire (structure à deux dimensions). Un gestionnaire de SPL peut vouloir imposer une structure hiérarchique particulière. Par exemple, « *Hosting* below *General* » à la ligne 14, signifie que la vue *Hosting* est la fille de *General*. Cette instruction est prise en compte par le convertisseur et la hiérarchie du FM résultant sera adaptée en conséquence. La description complète du principe des sous-vue est développée à la Section 5.1.1.

4.3.3 Étapes de conversion

Comme énoncé dans l'introduction de ce chapitre, notre convertisseur prend en entrée un modèle de conversion. Celui-ci contient, de façon normalisée, la variabilité du catalogue de produit associé. Le processus d'extraction consiste en 3 étapes décrites dans les Sections 4.3.3, 4.3.3 et 4.3.3. Pour illustrer le fonctionnement de notre processus, nous nous baserons sur l'exemple Wikimatrix précédemment introduit au Chapitre 3. Nous avons créé un modèle de conversion contenant trois fichiers de données qui ont été obtenus en filtrant avec notre DSL les données collectées. Nous avons juste gardé les caractéristiques et les entités qui nous semblaient utiles et nous les avons distribuées en 3 vues : *General*, *Functionnalities* et *Requirements* représentées respectivement par les Figures 3.1, 3.2 et 3.3. Les directives permettant de générer ce modèle de conversion sont décrites à la Section 7.3.

Notre approche prend FAMILIAR comme langage de manipulation de FM car il permet d'illustrer nos propos. Cependant, l'utilisateur pourra utiliser n'importe quel autre langage de manipulation de FM tant que celui-ci gère les opérations de fusion, d'insertion et les boucles.

Génération des FMs représentant l'association d'une Vue et d'une Entité

Cette étape consiste en la génération d'un FM pour chaque entité couplée à une vue. Chacun de ces FMs contient la variabilité associée à une rangée d'une vue particulière. Il y a donc autant de FMs généré que l'union de toutes des rangées des différents tableaux du modèle de conversion. Par exemple, les Figures 4.4, 4.5 et 4.6 décrivent visuellement la variabilité de *PBwiki* des Figures 3.1, 3.2 et 3.3. Ces FMs générés sont stockés dans des

de ces facilités dans le DSL en ne prenant que les pertinentes dans le cadre dans l'ingénierie des SPLs et la modélisation de features.

variables nommées par convention : ***ID_ NomDeVue***, avec ***ID*** le nom de l'entité et ***NomDeVue***, le nom de la vue associée à cette rangée. Le nom de ces variables est très important pour l'étape de transformation suivante. En effet, grâce à une expression régulière, il sera facile de manipuler l'ensemble des FMs pour une entité particulière (***ID_ ****). Le nom de leur racine est le nom de la vue associée. Chacun de ces FMs consiste en un niveau de vue pour une entité particulière.

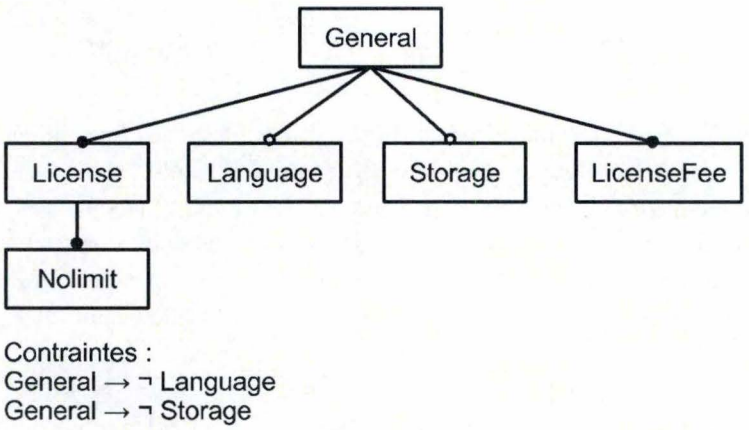

FIGURE 4.4 – FM ***PBwiki_ General***

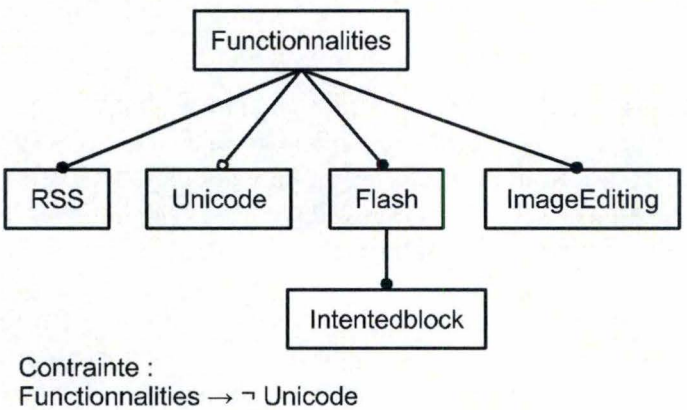

FIGURE 4.5 – FM ***PBwiki_ Functionnalities***

Le code FAMILIAR du Listing 4.2 place dans la variable ***PBwiki_ General*** un nouveau FM correspondant à celui de la Figure 4.4. Un FM est composé d'un ensemble de productions et de contraintes. Une production définit une relation entre une feature parente et les features filles. La ligne 2 donne

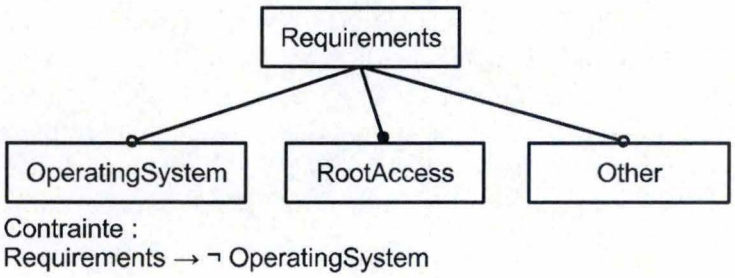

FIGURE 4.6 – FM ***PBwiki_Requirements***

General comme parent et les caractéristiques License, ProgrammingLanguage, DataStorage et LicenseCost_Fee dans un *groupe Et*. Les features délimitées par des crochets sont *Optionnelles*. La ligne 3 indique que NoLimit est une feature fille obligatoire de License. Les lignes 4 et 5 sont les deux contraintes associées au FM. Elles impliquent que si la feature *General* est sélectionnée, alors les features ProgrammingLanguage et DataStorage sont déselectionnées.

```
1 PBwiki_General = FM (
2    General : License [ProgrammingLanguage] [DataStorage]
       LicenseCost_Fee ;
3    License : Nolimit ;
4    General -> !ProgrammingLanguage;
5    General -> !DataStorage;  )
```

Listing 4.2 – Équivalent en FAMILIAR de la Figure 4.4

Comme expliquée ci-dessus, la présente étape dans le processus d'extraction génère un FM par rangée du modèle de conversion. Ce FM est constitué de structures résultant de l'interprétation des différentes valeurs de chacune de ces rangées. Voici les différentes valeurs prises en charge et leur traduction en terme de features et contraintes :

- Si la valeur équivaut à « Yes », nous créons une feature *Obligatoire* avec le nom de la caractéristique associée. Dans la Figure 4.4, la caractéristique LicenseFee est traduite en une feature du même nom et est rendue *Obligatoire* car la valeur associée vaut « Yes ».
- Si la valeur équivaut à « No », nous créons une feature morte avec le nom de la caractéristique associée. Cela signifie que nous créons une feature optionnelle avec une contrainte qui la rend non sélectionnable. Cette construction montre au configurateur de la SPL que cette feature est non gérée par le produit sélectionné. Dans la Figure 4.4, Language est une feature morte. En effet, elle est rendue *Optionnelle* et il lui est adjoint la contrainte « $General \rightarrow \neg Language$ » car la valeur associée

est « No ».
- Si la valeur équivaut à « N_A », cela signifie qu'il n'y a pas de valeur connue. En considérant que l'on ne sait pas si cela signifie que la caractéristique est sélectionnable ou pas, nous générons une feature *Optionnelle*. Dans la Figure 4.6, la feature Other est rendue *Optionnelle* car la valeur associée est « N_A ».
- Si la valeur est une valeur textuelle quelconque (autre que les trois précédemment énoncées), nous créons une feature *Obligatoire* avec le nom de la caractéristique associée et nous lui adjoignons une feature fille. Cet enfant sera nommé par la valeur textuelle. Dans la Figure 4.4, la feature *Obligatoire* License contient une sous-feature *Obligatoire* nommée Nolimit. « Nolimit » est la valeur textuelle associée à License.

Les FMs obtenus sont nommés « FM d'étape 1 » ou « FMs d'Entité x Vue ».

Génération des FMs d'entités

Une fois que l'entièreté de la variabilité a été convertie en un certain nombre de FM d'étape 1, nous agrégeons ceux-ci en les groupant en de nouveaux FM en fonction de leur nom d'entité. Chacun des FMs qui résultent de cette étape contient l'entièreté de la variabilité associée à une entité. Nous les nommons « FMs d'étape 2 » ou bien « FMs d'entité ». Ils sont placés dans des variables nommées par convention ***fm_ NomEntite*** avec ***NomEntite***, le nom de l'entité concernée. Nous obtenons autant de FMs d'étape 2 que d'entités présentes dans le modèle de conversion.

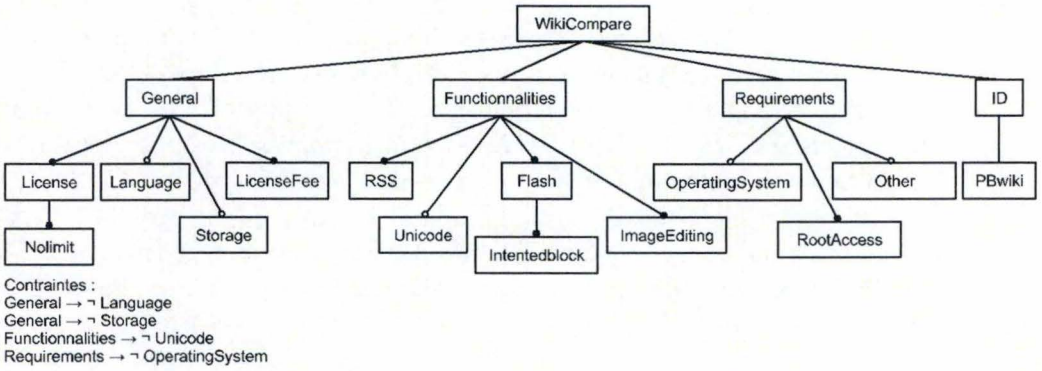

FIGURE 4.7 – ***fm_ PBwiki***

Les FMs produits par cette étape ont comme feature racine RootFeatureName, un nom donné par l'utilisateur. Celui-ci est le premier point nécessaire

à l'opérateur de fusion à l'étape suivante du processus d'extraction. Ce nom particulier représente le FM à produire. Dans le cas de Wikimatrix, nous avons décidé de le nommer WikiCompare, car le FM final représente les données de comparaison des moteurs de fusion.

Pour grouper la variabilité de l'ensemble des FMs de l'étape 1 selon une entité particulière, il suffit de les insérer juste en dessous de la racine. Ensuite, on associe l'identifiant de l'entité à ce FM. Pour ce faire, nous créons une feature obligatoire nommée par convention ID et qui est placée en dessous de la racine. Nous allons adjoindre une feature fille *Obligatoire* à celle-ci qui contient l'identifiant de l'entité associé à ce FM.

La Figure 4.7 nous donne une illustration graphique du FM produit à cette étape pour l'entité *PBwiki*. Les sous-arbres de ce FM, ayant comme racine Functionalities, Requirements et General, placés juste en dessous de la racine viennent des FMs de l'étape 1 et correspondent aux différentes vues associées à l'entité concernée. Elles sont illustrées par les Figures 4.4, 4.5 et 4.6. Ces vues sont des points pivots de la fusion à l'étape suivante.

```
1 fm_ENTITYNAME = FM(RootFeatureName : ID ; ID : ENTITYNAME ; )
2 foreach(f in ENTITYNAME*) do
3 insert f into fm_ENTITYNAME.RootFeatureName with mand
4 end
```

Listing 4.3 – Script de génération du FM d'entité de ENTITYNAME

Pour obtenir les FMs d'entité, il faut exécuter le script FAMILIAR du Listing 4.3 pour chacune des entités. Ce script va créer un squelette de FM composé du RootFeatureName avec comme enfant la feature ID ayant elle aussi une sous feature ayant le même nom que l'entité concernée, c'est-à-dire ***ENTITYNAME*** (ligne 1). Le fait de placer ce FM dans une variable nommée par convention ***fm_ENTITYNAME*** est très important. En effet, par après, il sera facile de les récupérer avec l'expression régulière ***fm_*** *. Ce script va boucler pour chacun des FMs de l'étape 1 (lignes 2 et 4). En effet, ceux-ci seront mis tour après tour dans la variable temporaire f (***f in ENTITYNAME***). Chacun de ces FMs, sera alors inséré en dessous de la racine de ***fm_ENTITYNAME*** (ligne 3). La racine du FM inséré aura une *relation Obligatoire* avec son nouveau parent.

Génération du FM final

À cette étape, l'algorithme a un ensemble de FM d'entité et doit les combiner en un FM qui contiendra la variabilité de l'entièreté du modèle de conversion. Grâce aux concepts associés au modèle de conversion, il est

plus que probable que ces FMs partagent un grand nombre de features communes (exemple : la feature racine, la feature spéciale ID, les différents noms de vues, les noms des caractéristiques,...). Dans ce cas, l'opérateur de fusion peut être utilisé pour fusionner les parties des différents FMs qui se chevauchent et donc créer un FM compact qui représente exactement l'union des configurations des FMs d'entités. L'opérateur utilisé pour la fusion se base sur une association par nom de feature : deux features correspondent si et seulement si elles ont le même nom. Plusieurs modes de fusion existent comme l'intersection, l'union, etc. Avec l'*union stricte*, nous voulons obtenir un nouveau FM dont chaque configuration valide est une configuration valide de FM_1 ou FM_2, ou n'est pas une configuration d'aucun des deux FMs.

$$[\![FM_1]\!] \cup [\![FM_2]\!] \neq [\![Result]\!]$$

Comme la fusion dans son mode d'union stricte est commutative et associative, plus de deux FMs peuvent être fusionnés. Avec cet opérateur utile, l'obtention du FM compact résultant du processus d'extraction s'effectue en exécutant une simple commande FAMILIAR donnée par le Listing4.4.

```
1 finalFM = merge sunion fm_*
```

Listing 4.4 – Fusion des FMs d'entité

L'expression régulière ***fm_ **** se réfère à tous les FM d'entités de l'étape précédente. Par définition de l'opérateur de fusion, ***finalFM*** contient l'ensemble de toutes les configurations de toutes les entités et garde bien évidemment le lien entre les caractéristiques et l'identifiant. Les identifiants sous la feature réservée ID font donc le lien entre l'ensemble des configurations associées et l'entité. Ce lien est très pratique dans le cas d'un configurateur. En effet, le gestionnaire de SPL, pour sélectionner le système de wiki qui lui convient, sélectionnera certaines des features qui correspondent à ses besoins. Il voudra alors savoir quel est le système de wiki qui comprenant les feature sélectionnées. Pour ce faire, grâces aux contraintes induites par la fusion, la feature identifiante sera automatiquement sélectionnée par l'auto-propagation.

La Figure 4.8 montre le FM final généré avec notre processus d'extraction à partir des données des Figures 3.1, 3.2 et 3.3. Seulement un extrait des contraintes est renseigné par soucis de lisibilité.

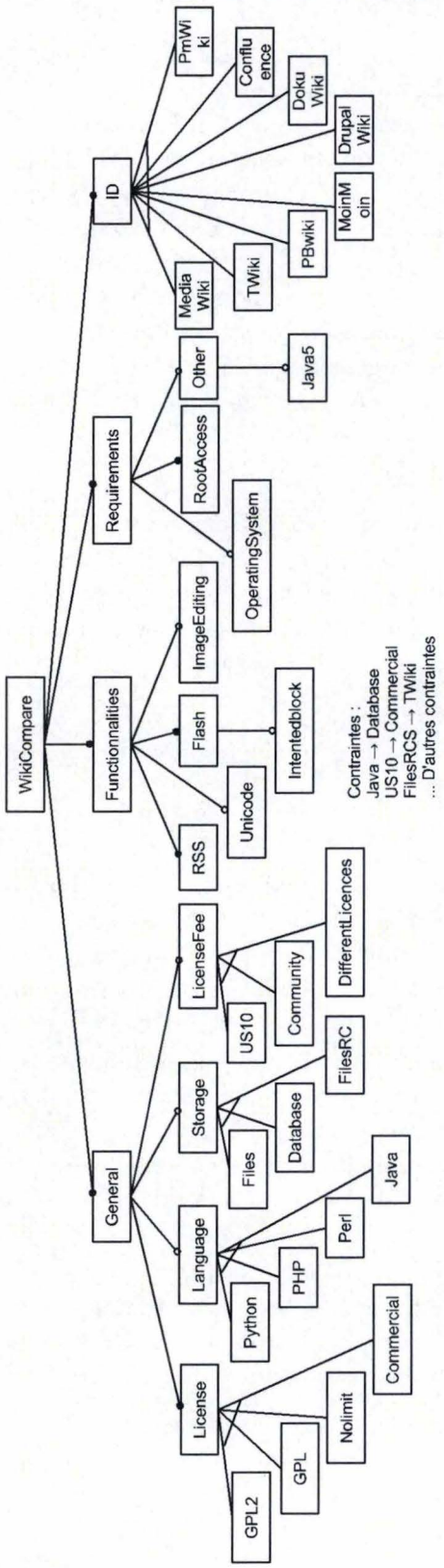

FIGURE 4.8 – ***finalFM***

Chapitre 5

Perfectionnement

Dans ce chapitre, des fonctionnalités supplémentaires sont introduites pour compléter notre processus d'extraction ainsi que l'utilisabilité de FAMILIAR. La Section 5.1 parle d'extensions à notre approche. La Section 5.2 concerne la création d'un pont entre le formalisme TVL et FAMILIAR.

5.1 Perfectionnement du processus d'extraction

Vu que notre approche se veut générique, nous introduisons des extensions au processus d'extraction pour donner un certain degré de liberté supplémentaire à l'utilisateur. Les directives de structuration de vues, les identifiants composés [1] et les champs multivalués sont présentés respectivement dans les sections 5.1.1, 5.1.2 et 5.1.3. La combinaison du paramétrage, du filtrage, du modèle de conversion avec les concepts qu'il apporte, des extensions au modèle présentées dans cette section ainsi que la souplesse de FAMILIAR (grâces à ses opérations de manipulation de FM) permettra au manager de SPL d'extraire automatiquement et facilement le FM ayant la structure et le contenu adéquat d'un catalogue de produit existant.

5.1.1 Structuration de vues

Le FM résultant de l'extraction peut manquer de granularité au niveau de la variabilité qu'il représente. En effet, actuellement, il n'existe que trois niveaux de détails : que sont les *vues*, les *caractéristiques* et les *valeurs*, définies à la Section 4.3.1 et illustrées par la Figure 4.8. Il pourrait donc être intéressant de combiner les vues pour créer une hiérarchie dont chaque niveau est de plus en plus détaillé. Notre approche donne la possibilité de (1) créer des vues dites « techniques », c'est-à-dire des vues qui ne contiennent pas de caractéristiques et qui ne sont pas associées à des tableaux dans le modèle de conversion. Elles permettent donc de créer des niveaux de détails

1. Un peu au même titre que dans les bases de données.

et donc de généraliser ou abstraire d'autres vues ; (2) placer des vues en dessous d'autres. Nous les nommons « sous-vue ».

Pour donner encore plus de facilités au gestionnaire de SPL pour manipuler ses vues et la structure finale du FM à extraire, nous avons développé une opération supplémentaire. Elle consiste en l'agrégation de deux vues. Le résultat de cette agrégation est la juxtaposition du contenu de la première vue à la seconde. La seconde vue contiendra alors toute la variabilité des deux vues originales.

La création de cette extension a été motivée par le cas d'utilisation présenté dans [12]. Il concerne l'extraction de la variabilité relative à la configuration du noyau Linux et indique que la profondeur maximum du FM est de 8 niveaux. Il est donc indispensable de fournir à l'utilisateur le concept de sous-vue pour dépasser les limites initiales de notre approche.

Des directives du DSL permettent à l'utilisateur de renseigner la structure du FM à extraire, nous les énonçons avec un petit exemple :

```
1  structure:
2      hardware below computer
3      // hardware est une sous-vue de computer
4      software abstract OS apps
5      // software est une nouvelle vue technique ayant comme
6      // sous-vues OS et apps
7      software below computer
8      // software est une sous-vue de computer
9      memory moveIn hardware
10     // hardware contient la variabilité de memory
```

Listing 5.1 – Exemple de déclaration de la structure de vues

Comme présenté dans le Listing 5.1, trois instructions existent : (1) « vue1 below vue2 » place la vue1 en dessous de la vue2 ; (2) « vue1 abstract vue2 » crée la vue vide vue1 et place vue2 en dessous de celle-ci ; (3) « vue1 moveIn vue2 » copie le contenu de vue1 et l'ajoute à la vue vue2.

Il est important de noter que les directives de structuration seront opérées par ordre d'écriture. Par exemple :

```
1  structure:
2      view1 moveIn view2
3      view2 moveIn view3
```

Listing 5.2 – Exemple de création de sous-vue : Ordre 1

N'a pas le même effet que :

```
1 structure:
2     view2 moveIn view3
3     view1 moveIn view2
```

Listing 5.3 – Exemple de création de sous-vue : Ordre 2

En effet, dans le Listing 5.2, view3 contient à la fois view2 et view1. Par contre, dans le Listing 5.3, view3 ne contient que view2.

5.1.2 Identifiants composés

Pour introduire le concept d'identifiants composés, nous utiliserons un scénario typique : un configurateur d'ordinateurs portables. Le gestionnaire de la SPL doit attribuer un identifiant unique pour chacun de ses produits. Cependant, cet identifiant peut manquer de granularité. En effet, il pourrait être intéressant de distinguer les ordinateurs portables de gammes ou de marques différentes. Ces critères de choix peuvent être très pratiques dans le processus de configuration. L'utilisateur configurant la SPL veut, par exemple, obtenir la liste des produits qui ont une marque et une quantité de mémoire vive particulière. Pour ce faire, il sélectionne les deux features de son choix et demande au moteur de raisonnement de lui donner la liste des configurations valides par rapport à la configuration actuelle. Il peut alors effectuer son achat à partir des différents produits proposés. Dans la pratique, cette fonctionnalité est recherchée, car très utilisée.

Dans le cas où les identifiants composés ne sont pas gérés, il suffit au gestionnaire de la SPL d'ajouter les caractéristiques Gamme et Marque à chacune des vues du modèle de conversion. Cette façon de faire amène un certain nombre de problèmes : (1) il se peut que les valeurs associées aux caractéristiques Gamme et Marque varient pour une même entité selon les vues. En effet, il n'y a pas de contraintes d'unicité sur la composition des 3 identifiants ; (2) ajouter les caractéristiques Gamme et Marque sur toutes les vues amène à des duplications d'informations assez importantes sur le FM final. Pour répondre à cette problématique, nous avons décidé d'ajouter une extension permettant la composition d'identifiants à notre approche.

Notre convertisseur propose au gestionnaire de SPL de définir des clés secondaires[2]. Le n-uple de clés secondaire doit être unique pour une même entité. Les clés sont définies grâce au DSL via la directive :

2. Par opposition à la clé primaire, dont les valeurs identifient les entités.

```
default:
    parsing:
        key: "Identifiant" "Marque" "Gamme"
        // première clé: primaire
        // clés suivantes: clés secondaires
```

Listing 5.4 – Exemple identifiant composé

À la place de dupliquer les clés secondaires à travers tout l'arbre dans les vues associées, nous avons décidé de les placer à l'endroit où elles ont le plus de sens, c'est-à-dire sous la feature réservée ID. Nous y reconstituons une hiérarchie de clés définies par l'ordre de définition dans la directive du DSL. Au sommet (c'est-à-dire juste en dessous de ID), sera placée la première clé secondaire. Au second niveau (c'est-à-dire juste en dessous des premières clés secondaires) seront placés les deuxièmes, et ainsi de suite jusqu'à ce qu'il n'y ait plus de clés secondaires. On mettra alors les clés primaires comme feuilles de l'arbre ainsi composé. Le FM résultant de l'import du catalogue de produit sera alors cohérent au point de vue des identifiants et l'utilisateur n'aura aucun mal a le configurer à sa guise.

5.1.3 Valeur de type multivaluée

Comme décrite à la Section 4.3.1, une valeur du modèle de conversion peut être de plusieurs types. Elle peut être soit textuelle, soit particulière et donc induire une structure prédéfinie dans le FM converti (*Obligatoire*, *Optionnelle* ou *feature morte*) ou encore être multivaluée. Techniquement, une valeur multivaluée est une valeur dont le contenu a un séparateur particulier.

```
1 default:
2     parsing:
3         separator: ","
4         multivalued_separator: ";"
```

Listing 5.5 – Définition du séparateur des valeurs multivaluées

Ce nouveau séparateur est défini par les directives du Listing 5.5. La ligne 3 correspond à la définition de la virgule comme séparateur des valeurs et la ligne 4 à la définition du point-virgule comme séparateur de valeurs multivaluées.

```
1 Identifier ,OS_Comp, License
2 Gimp, Linux ;MacOS; Windows , Free
3 Photoshop , Windows ; MacOS, Shareware
4 Paint . net , Windows , Free
```

Listing 5.6 – Exemple CSV multivalué

Le Listing 5.6 nous montre un exemple de fichier CSV donné en entrée à notre convertisseur. Celui-ci contient des données relatives à des programmes de retouche d'image. Pour une entité donnée, la caractéristique OS_Comp représente la compatibilité du logiciel avec un ou plusieurs systèmes d'exploitation. Certaines des valeurs associées à cette caractéristique sont multivaluées. Par exemple, l'entité *Gimp* est compatible avec les systèmes Linux, MacOs et Windows tandis que *Paint.net* est uniquement compatible avec Windows. Comme montré dans cet exemple, notre convertisseur permet de combiner les valeurs normales et multivaluées pour une même caractéristique.

La traduction d'une valeur multivaluée en une structure de FM est s'effectue comme suit : (1) en dessous de la feature représentant la vue, nous créons une feature ayant le même nom que la caractéristique ; (2) nous parcourons les différentes valeurs multivaluées. S'il existe des valeurs textuelles, nous créons un sous-groupe contenant autant de feature que de valeur textuelle. Ce groupe est par défaut un *groupe Alternatif* mais celui-ci peut être interprété différemment selon les directives de l'utilisateur. Le Listing 5.7 donne un exemple d'interprétation de valeurs multivaluées. Tous les groupements gérés par FAMILIAR sont utilisables ; (3) si une valeur représentant une feature morte existe, nous créons la contrainte associée ; (4) nous inférons les relations si des valeurs particulières sont combinées.

```
1 // Définition de l'interprétation par défaut
2 default:
3    multivalues: Alternatives
4
5 view: general
6    // Surcharge de l'interprétation des valeurs multivaluées
        pour des caractéristiques données
7    multivalues:
8       "Other Limits" => Alternatives
9       "Intended Audience" => OR-Alternatives
```

Listing 5.7 – Définition de l'interprétation des valeurs multivaluées

Pour illustrer le processus de conversion de valeurs multivaluées par un exemple, nous donnons le Listing 5.8 à notre convertisseur, qui produit la Figure 5.1. La combinaison de plusieurs valeurs textuelles avec la valeur particulière *Optionnelle* implique que la feature Caractéristique est optionnelle et a un *sous-groupe Alternatif* contenant les valeurs textuelles Val_Textuelle1 et Val_Textuelle2. Si un utilisateur sélectionne la caractéristique, il sera alors obligé de sélectionner une des sous-features pour obtenir une configuration valide.

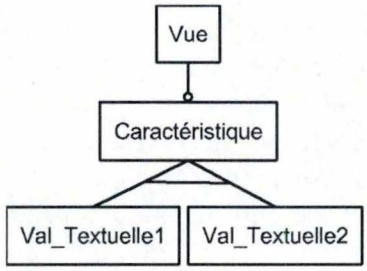

FIGURE 5.1 – Combinaison valeurs multivaluées textuelles avec optionnelle

```
1 Identifier , Caractéristique
2 ID , Opt; Val_Textuelle1 ; Val_Textuelle2
```

Listing 5.8 – Exemple CSV combinant plusieurs types de valeurs multivaluées

5.2 Pont TVL vers FAMILIAR

Dans cette section, en plus du processus d'extraction, une contribution supplémentaire a été apportée au mémoire. En effet, les implémentations de programme de gestion de FMs ont des caractéristiques propres qui font leurs forces et faiblesses. Ce mémoire se base principalement sur FAMILIAR [30] car celui-ci a été développé en vue de manipuler et raisonner de manière programmative à propos des FMs. Il fournit aussi des opérateurs indispensables au processus d'extraction. Cependant, celui-ci n'apporte pas certaines facilités pour la modélisation manuelle d'une SPL. En effet, les constructions prises en charge sont assez basiques. Il est, par exemple, très difficile de définir des *groupes à Cardinalité* (énoncées à la section 2.1), impossible de donner à features différentes le même nom ou encore, d'utiliser des mécanismes de modularisation [13]. Nous avons donc développé un pont entre TVL et FAMILIAR, présentés à la Section 2.3.2, pour permettre à l'utilisateur de modéliser un FM avec les richesses syntaxiques de TVL et de, par la suite, le manipuler avec FAMILIAR. De la même manière, l'utilisateur peut aussi, par exemple, extraire un FM à partir d'un catalogue de produit avec FAMILIAR pour, ensuite, le représenter avec la souplesse de TVL.

Comme énoncé dans le Chapitre 2, les concepts autour de la modélisation de SPLs ainsi que la manipulation de FM sont nombreux et beaucoup sont considérés comme des extensions des *FMs de base*. Chacun des programmes de gestion de FMs sur le marché fournit un sous-ensemble de ceux-ci dont certains sont sujets à des interprétations. C'est pourquoi, dans le cadre de création d'un pont entre TVL et FAMILIAR, nous comparons leur expressivité. Dans la Section 5.2.1, nous identifions tout d'abord l'ensemble des

fonctionnalités gérées par TVL et qui ne sont pas fournies par FAMILIAR. Nous donnons ensuite les différentes traductions ou interprétations de ces concepts dans le formalisme FAMILIAR. Finalement, nous donnons la liste des constructions non gérées par notre traducteur. Pour soutenir cette contribution, nous avons implémenté un outil permettant de traduire du TVL en FAMILIAR et inversement. L'implémentation ainsi que les tests unitaires pour la validation de celui-ci sont décrits à la section 5.2.2. Celle-ci s'intéresse uniquement à la représentation interne normalisée de TVL et en crée directement la représentation FAMILIAR associée en terme d'objet. Ces deux langages de modélisation ne sont donc pas manipulés textuellement.

5.2.1 Équivalence de constructions

TVL donne des facilités pour modéliser de manière intuitive le FM. Beaucoup de constructions sémantiques sont donc fournies, dont certaines peuvent être qualifiées de sucre syntaxique. Pour rester interprétable en logique booléenne, il a été nécessaire de transformer le langage TVL, très expressif, en un formalisme normalisé beaucoup plus basique. Ce formalisme a pour ainsi dire la même expressivité que FAMILIAR, qui modélise les *FM de base.*

Faute de temps, de faisabilité et d'utilité, certaines constructions de TVL ont été mises de côté. Seules les constructions qui semblaient les plus pratiques et une interprétation presque directe ont été intégrées à l'outil de traduction. Nous avons sélectionné les fonctionnalités générant un FM FAMILIAR qui reste intuitif et nous avons estimé que les opérations traduites étaient suffisantes pour créer des FMs expressifs. Les opérations non traduites sont simplement ignorées par l'outil de traduction et signalées à l'utilisateur.

Relations et groupes

Comme expliqué dans la Section 2.1, un groupe à cardinalité correspond à une intervalle dénotée par `<n..m>`, avec $n \leq m$, qui limite le nombre de feature fille qui peut faire partie d'un produit lorsque la feature parente est sélectionnée [11]. Les cardinalités sont utilisées par TVL pour tout type de groupement. En effet, les intervalles `<N..N>`, `<1..N>`, `<1..1>` correspondent respectivement aux *groupements Et*, *Ou* et *Alternatif* avec N le nombre de features filles. Celles-ci sont directement transposables en FAMILIAR. Cependant, les intervalles avec des cardinalités autres que les 3 identifiées ci-dessus ne sont pas intuitifs à simplifier en un *FM de base.* En effet, il est nécessaire d'ajouter des contraintes qui réduisent la liste des configurations de la même manière que la cardinalité associée. Nous nous basons sur l'approche décrite par [50], qui génère une contrainte optimisée utilisant des variables temporaires en fonction des bornes n et m. Cette fonction est notée dans la littérature $card[i..j]$.

Étapes du processus de traduction :
- Nous créons une liste de production[3] FAMILIAR dénotée *ProdL*.
- Nous créons une liste de contraintes[4] FAMILIAR dénotée *ContL*.
- L'algorithme suivant est lancé à partir d'une feature particulière. Celui-ci est récursif et il est convenu de lui donner la référence de la feature racine.
  - Récupération la liste des features filles.
  - Détermination de leur nombre dans la variable N.
  - Si $N = 0$, création d'une production avec la feature donnée en paramètre, stop.
  - Sinon, identification de la relation de groupe associée.
    - Si $n = m = N$, création d'une production contenant un *groupe Et* avec les features filles et la placer dans *ProdL*. Compléter cette production par le caractère obligatoire ou optionnel de chacune des features filles.
    - Si $n = m = 1$, création d'un *groupe Alternatif* dans *ProdL*.
    - Si $n = 1$ et $m = N$, création d'un *groupe Ou* dans *ProdL*.
    - Sinon, création d'un *groupe Et* dans *ProdL* et ajout de la contrainte générée par la fonction $card[n, m]$ définie dans [50] à *ContL*.
  - Récupération des contraintes associées à la feature donnée en paramètre. Nous itérons pour chacune d'entre-elles :
    - Simplification de la contrainte par une fonction de l'API TVL. Celle-ci utilise uniquement les opérateurs $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$, $\neg$, (et).
    - Ajout de la contrainte à *ContL*.
  - Lancer récursivement cet algorithme pour chaque feature fille.
- Création d'un objet représentant un FM FAMILIAR qui contient *ProdL* et *ContL*.

Ajouter une contrainte visant à émuler une cardinalité particulière est la seule manière de traduire ce concept avancé en un *FM de base*. Le FM produit est donc différent sémantiquement (car il ne contient pas de relation de cardinalité) mais garde le même ensemble de configuration. De par sa complexité, la contrainte diminue l'expressivité et l'évolutivité du modèle, ce qui peut poser problème à l'utilisateur.

Cet algorithme constitue la base de tous les algorithmes décrits dans les sections suivantes, car celui-ci parcourt la structure complète du FM TVL

3. La structure interne de FAMILIAR se base sur des productions et des contraintes. Une production représente un niveau de relation entre les features. Par exemple, la production « A : (B|C|D) ; » (représentée ici de manière textuelle) dénote un *groupe Alternatif* composé des features B, C et D et ayant comme parent A.

4. En FAMILIAR, vu que les features ont des noms uniques, les contraintes ont une portée globale.

ainsi que ses contraintes.

Espace de nom

La gestion des noms de feature ne se fait pas de la même manière en TVL ou en FAMILIAR. En effet, contrairement à TVL, FAMILIAR n'autorise pas deux features différentes à avoir le même nom. Pour convertir un FM TVL contenant potentiellement des features de même nom, il a fallu introduire un espace de noms assez complexe.

Lors de la traduction, nous appliquons ce processus pour la gestion des noms :

- Pour chaque feature TVL rencontrée, nous opérons les sous-étapes suivantes :
  - Nous rendons le nom de feature compatible avec FAMILIAR en supprimant les espaces et autres caractères particuliers.
  - Nous vérifions si le nom de la feature n'a pas encore été utilisé en parcourant l'espace de nom.
    - Si oui, nous générons aléatoirement une particule que nous ajoutons au nom de la feature. Nous plaçons ce nom dans une variable prévue à cet effet et nous l'insérons dans l'espace de nom. Nous l'insérons aussi dans une table faisant la correspondances entre le nom d'origine et le nouveau nom généré.
    - Sinon, nous plaçons ce nom dans la variable prévue à cet effet, et nous l'insérons dans l'espace de nom.
  - Nous créons la feature FAMILIAR associée ayant le nom renseigné par la variable.
- Pour chaque contrainte contenues dans le FM TVL, nous opérons les sous-étapes suivantes :
  - Nous parcourons l'ensemble des références aux features.
    - Nous résolvons les noms[5].
    - Nous vérifions, grâce à la table des correspondance, si ceux-ci doivent être remplacé par le nom généré.
  - Nous ajoutons la contrainte au FM FAMILIAR.

Comme indiqué, nous utilisons des noms générés aléatoirement pour les features de même nom. La particule à ajouter au nom est une interprétation d'un concept non transposable en FAMILIAR. Il se peut que ce changement biaise l'expressivité du FM traduit. Il est, toutefois important de noter que le FM résultant conserve le même ensemble de configuration.

5. En effet, de par la non unicité des noms en TVL, ils sont qualifiés par un chemin relatif. Des accesseurs comme « this. », « parent. » ou encore « root. » sont fournis pour parcourir le FM.

Attributs

Contrairement au langage TVL décrit dans la littérature [16, 13, 17], son implémentation gère uniquement les attributs booléens. Leur traduction en un *FM de base* est relativement aisée. En effet, de par le caractère binaire d'une feature (sélectionnée - désélectionnée), celle-ci peut représenter un attribut booléen (contenant les valeurs vrai - faux). Pour ce faire, il suffit de placer en dessous de la feature concernée une nouvelle feature ayant le même nom que l'attribut booléen à traduire. Cependant, un problème de taille survient : FAMILIAR ne gère pas le multigroupe et ne permet donc pas d'ajouter une sous-feature (sous forme de *groupe Et*) s'il existe déjà un *groupe Alternatif* ou *Ou*. Dans ce cas, la solution envisagée est de créer une feature technique rendue obligatoire entre le parent et le groupe posant problème.

Voici les différentes étapes du processus de traduction des attributs qui est exécuté pour chacune des features du FM :

- La feature passée dans cet algorithme est dénotée $Parent$.
- Récupérer la liste des features filles dénotée ffL.
- Récupérer la liste les attributs dénotée laL.
- Si laL est vide, stop.
- Sinon :
  - Si le groupe identifié par ffL est un *groupe Ou*, *Alternatif* ou *à cardinalité*, créer une nouvelle production dénotée $nProd$ consistant en un *groupe Et* contenant une feature obligatoire avec un nom quelconque (généré aléatoirement, non encore utilisé). Modifier la production d'origine (définissant $Parent$) pour placer le groupe posant problème en dessous de la feature générée. Ajouter $nProd$ à la liste des productions du FM.
  - Si ffL est vide, créer une production contenant un *groupe Et* vide. Le placer dans la liste des productions du FM.
  - Si le groupe identifié par ffL est un *groupe Et*, ne rien faire.
  - Pour chaque attribut booléen, ajouter une feature obligatoire de même nom que l'attribut booléen en dessous de $Parent$.

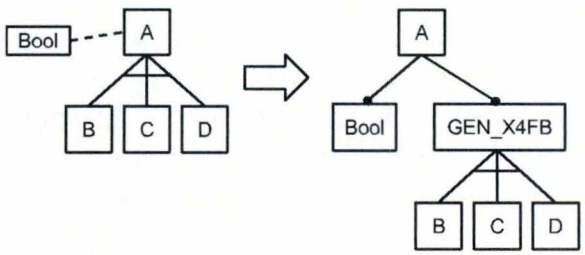

FIGURE 5.2 – Traduction attribut booléen avec sous-groupe Alternatif

La Figure 5.2 montre à gauche un FM TVL composé de 4 features (A, B, C, D), une racine (A), un *groupe Alternatif* (B, C, D) et d'un attribut booléen nommé *Bool*. À droite, nous pouvons voir le résultat de notre algorithme. Deux features sont placées en dessous de la racine A : la feature *Bool* qui est rendue obligatoire, ainsi qu'une feature technique nommée pour l'occasion GEN_X4FB. Le *groupe Alternatif* a été déplacé en dessous de la feature GEN_X4FB.

Lorsque l'utilisateur va lister les configurations du FM résultant, il va y trouver les features techniques. Nous lui conseillons de les filtrer (en fonction du préfixe) car elles entravent la lisibilité des configurations. Il faut toutefois noter que le nombre de configurations n'est pas modifié par celles-ci.

Énumération

L'énumération est un concept apporté par TVL. Déclarer un attribut comme étant du type *enum* signifie que l'attribut aura la valeur d'un des éléments d'un ensemble de valeurs prédéfinies. Par exemple, en TVL :

```
1 MoteurDeWiki{
2     enum Licence in {GPL, LGPL, MIT, Apache}
3 }
```

Listing 5.9 – TVL enum exemple

Ce code est la représentation d'une feature nommée MoteurDeWiki qui possède un attribut Licence de type enum pouvant prendre les valeurs « GPL », « LGPL », « MIT » ou « Apache ».

Le concept d'attribut, et donc du type enum, n'est pas présent en FAMILIAR. Pour le traduire, nous avons pensé en terme d'ensemble de configuration. L'utilisateur, en déterminant un attribut de type enum, veut restreindre le choix de la configuration à une seule des valeurs prédéfinies. Cela revient donc, en FAMILIAR, à créer un *groupe Alternatif* en dessous de MoteurDeWiki et composé de 4 features nommées GPL, LGPL, MIT et Apache.

De nouveau, vu que le multigroupe n'est pas géré par FAMILIAR, il faut utiliser des features techniques comme décrit à la section précédente lorsqu'il existe déjà des *groupes Ou*, *Alternatif* ou *à cardinalité*.

Constructions non traduites

Voici une liste des constructions non retenues pour la création du pont :

Graphe acyclique dirigé (DAG - Directed Acyclic Graph - une feature peut avoir de multiples parents) : Nous ne voyons pas une façon de réaliser cette traduction. De plus, cette construction est peu employée.

Attributs entiers Ceux-ci ne sont actuellement pas gérés par l'implémentation de TVL. De plus, il n'est pas possible avec l'implémentation FAMILIAR de raisonner sur les entiers, car cela nécessite un solveur particulier. De la même manière, les opérations sur les entiers ne sont pas gérées.

IfIn / IfOut Ces opérateurs sont traduits par l'utilisation de la fonction de simplification des contraintes fournie par l'API de TVL, et ce, uniquement dans le cas de valeurs booléennes. Ils ont un intérêt limité si les attributs numériques ne sont pas gérés car ils sont remplaçables par des contraintes.

Mécanisme de modularisation Celui de TVL est en partie interprété, mais non traduit. Celui-ci permet d'agréger la représentation d'un concept à un endroit unique, ce qui évite des duplications. Il permet, par exemple, de définir un sous-arbre ou une feature quelque part et, ensuite, de l'étendre une ou plusieurs fois à d'autres positions du FM. Il est donc possible, par exemple, de définir un type « Ordinateur » qui est composé des features Processeur, Memoire, Hd, etc. et de l'étendre à un ensemble de gammes de produits. Deux personnes peuvent travailler sur le FM sans altérer le travail de l'autre. Il est possible d'ajouter des contraintes supplémentaires sur les « instances ».

Pour notre algorithme, nous nous sommes basés sur une représentation interne normalisée de TVL et non pas sur le modèle textuel. Dans le modèle normalisé, les différents modules sont déjà assemblés et d'autres constructions y sont simplifiées. La façon de définir un FM par module est difficilement émulable par FAMILIAR. En effet, il faudrait utiliser une multitude de FM (pour chaque module ou concept) et les insérer à la volée dans le FM à produire. Cette opération amènerait des problèmes de gestion de noms dupliqués.

Les différentes constructions non traduites de cette section sont soit des fonctionnalités mineures, soit des fonctionnalités mal gérées par TVL. Nous pouvons donc dire que l'ensemble des constructions traduites est suffisant pour la majeure partie des modélisations en TVL.

5.2.2 Implémentation et tests

Une implémentation du pont TVL vers FAMILIAR a été réalisée[6]. Elle est centrée sur la traduction des représentations internes Java de ces deux gestionnaires de FMs. Des tests unitaires ont été écrits pour vérifier que les constructions traduites correspondent à l'entrée et pour valider les deux implémentations en vérifiant l'ensemble des configurations. La comparaison des configurations est une tâche qui a été bénéfique pour les 3 projets, c'est-à-dire à TVL, FAMILIAR et au pont entre ces deux formalismes. En effet, cela a permis d'ajuster la traduction du pont TVL ainsi que de déceler des bugs ou des interprétations conflictuelles des FMs entre FAMILIAR et TVL.

6. Le code source du projet est disponible en ligne sous forme d'archive zip à l'adresse `http://www.douby.be/dl/realisations/memoire/TVLTranslator.zip`. Il est aussi disponible sur le serveur Subversion `https://www.webkot.be/svn/tvltranslator/`. L'utilisateur veillera, pour se connecter, à utiliser l'identifiant « guest » et le mot de passe « guest »

Chapitre 6

Architecture et implémentation

Nous donnons, dans ce chapitre, une ébauche de l'architecture et des choix d'implémentation de notre approche. Dans la Section 6.1 nous donnerons l'architecture générale de notre approche en l'illustrant par un diagramme de classe. Nous expliquerons alors les choix technologiques et architecturaux qui ont guidé l'implémentation dans la Section 6.2. Nous y dressons aussi un bref aperçu des tests unitaires ayant pour but de valider les fonctionnalités ainsi que la justesse des FMs produits par notre outil. Pour finir, dans la Section 6.3, nous indiquons à l'utilisateur comment celui-ci peut récupérer le code source du projet.

6.1 Architecture générale

Pour décrire l'architecture générale de notre approche, nous utilisons le diagramme de classe de la Figure 6.1.

Notre convertisseur utilise un certain nombre de classes :

FMLConvert Cette classe opère les 3 étapes du processus de conversion sur l'objet ConversionModel passé en paramètre pour obtenir le script FAMILIAR intermédiaire ou le FM final. Si l'utilisateur le souhaite, elle crée l'objet Skeleton associé pour l'optimisation par structure approximée. Un ensemble de méthodes est fourni pour récupérer des métriques et déterminer les valeurs textuelles qui sont interprétées en terme de structures particulières.

ConversionModel Cette classe représente le modèle de conversion. Elle contient la variabilité de la SPL associée et fourni des méthodes pour filtrer et adapter les données aux besoins de l'utilisateur. Il est, par exemple, possible de filtrer des vues, des caractéristiques, de garder un certain nombre d'entités,...

ConversionModelArray Constitue une vue du modèle de conversion. Elle est associée à un objet ConversionModel et peut avoir un ensemble de

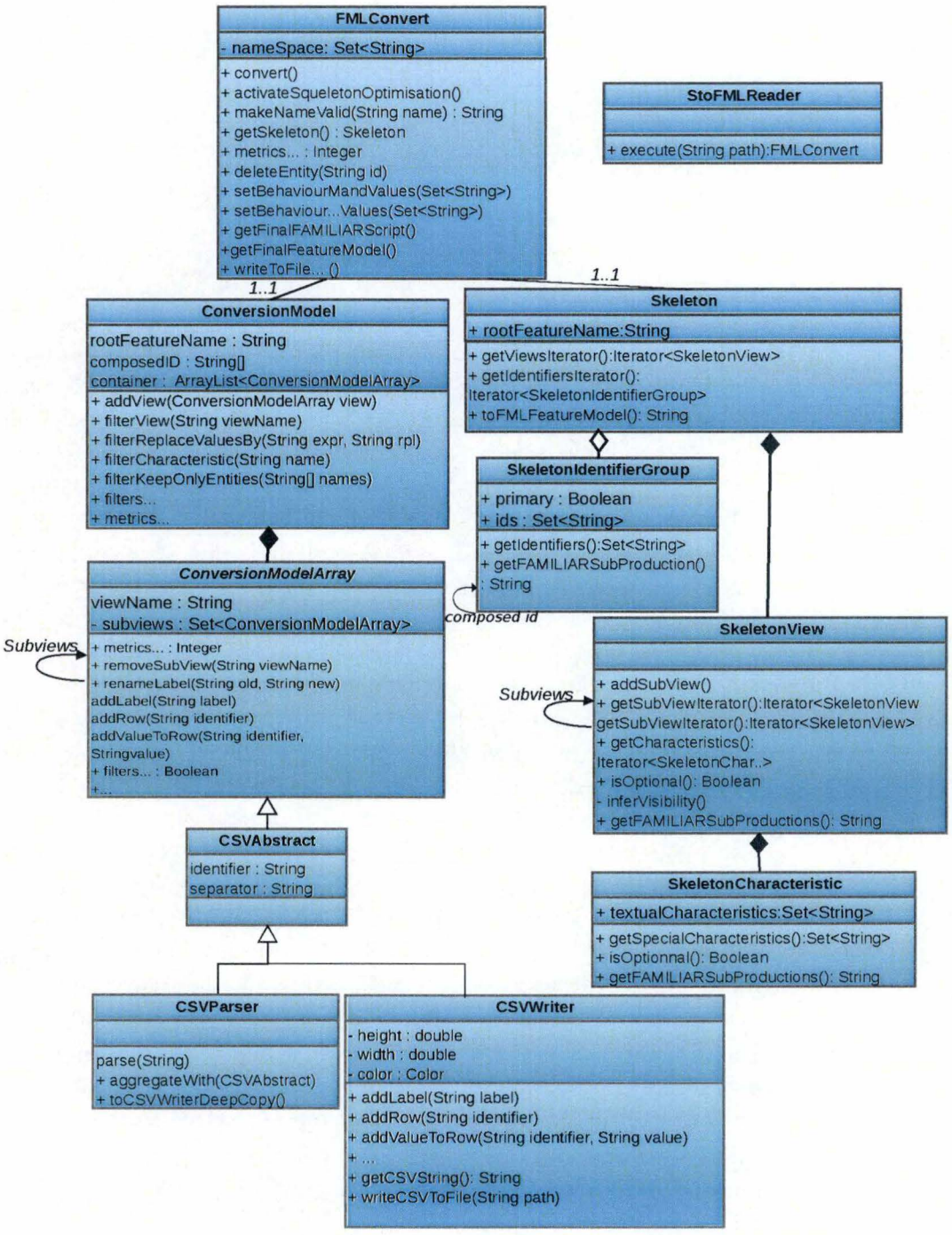

FIGURE 6.1 – Diaramme de classe de l'implémentation de notre contribution

sous-vues. Des méthodes sont fournies pour manipuler la structure en tableau, c'est-à-dire, modifier l'en-tête, ajouter des labels, des rangées, des valeurs etc. Un ensemble de fonctions fournies permettent de filtrer la vue.

CSVAbstract Abstrait les fichiers CSV et regroupe certaines méthodes privées et des attributs. Cette classe hérite de ConversionModelArray.

CSVParser Cette classe hérite de CSVAbstract et donc de ConversionModelArray. Elle est un accesseur aux fichiers CSV car elle implémente un parseur. Elle fournit des méthodes pour, agréger le fichier CSV parsé avec un autre fichier et convertir cet objet en un objet du type CSVWriter.

CSVWriter Cette classe hérite de CSVAbstract et donc de ConversionModelArray. Elle permet de créer manuellement un fichier CSV et donc le tableau du modèle de conversion associé. Des méthodes sont fournies pour, par exemple, ajouter une rangée, un label, une valeur, etc. Elle donne la possibilité, une fois le tableau CSV constitué, de récupérer le code CSV associé ou de le placer dans un fichier.

Skeleton Est la représentation objet de la structure approximée permettant une optimisation de la performance du processus de conversion. Elle permet de générer, avec les structures d'objets sous-jacentes, la représentation textuelle FAMILIAR de la structure approximée. Des méthodes sont fournies pour récupérer la liste des identifiants et des vues.

SkeletonIdentifierGroup Est la représentation d'un niveau d'identifiant. Nous parlons de niveau, en référence aux identifiants composés, présentés comme extension du processus d'extraction à la Section 5.1.2. Cette classe fournit une méthode pour obtenir les productions FAMILIAR associées aux identifiants sous leur représentation textuelle.

SkeletonView Est la représentation d'une vue pour la structure approximée. Elle gère les sous-vues décrites à la Section 5.1.1. Des méthodes permettent d'inférer sa visibilité selon les critères énoncés par l'algorithme de la Section 8.3.1. Une méthode est fournie pour obtenir les productions FAMILIAR associées dans leur représentation textuelle.

SkeletonCharacteristic Est la représentation d'une caractéristique pour la structure approximée. Une méthode est fournie pour obtenir les productions FAMILIAR associées dans leur représentation textuelle.

StoFMLReader Permet d'obtenir, avec un fichier de directive passé paramètre, un objet FMLConvert qui contient un modèle de conversion complet, paramétré et filtré.

L'utilisation de l'héritage entre CSVAbstract et ConversionModelArray n'est pas anodin. En effet, ce découpage en classe rend notre approche plus générique. Pour l'import de données provenant d'un autre type de fichier que

CSV, il suffit, à l'utilisateur, d'étendre ConversionModelArray pour créer un accesseur aux données de son choix.

Un effort particulier à été investi dans cette implémentation pour qu'elle soit rigoureuse, performante et générique. De plus, des facilités sont données à l'utilisateur pour prendre en main facilement cet outil. Nous avons donc (1) mis sur pied une documentation ; (2) développé le code autour de design patterns ; (3) développé une structure d'objets pour que l'utilisateur puisse facilement transformer ses données en entrée en un modèle de conversion ; (4) établi un ensemble conséquent de tests unitaires ; (5) crée un DSL pour faciliter la paramétrage du convertisseur, le filtrage de l'entrée et la structuration du FM résultant.

Comme la description des classes l'a sous-entendu, le DSL que nous fournissons à l'utilisateur consiste uniquement en des appels de méthodes de filtrage des classes FMLConvert, ConversionModel et ConversionModelArray. Les appels aux méthodes de ConversionModelArray ont une portée locale à la vue associée, ceux de ConversionModel ont une portée globale, sur l'ensemble des vues et des données. Finalement, les appels à FMLConvert permettent de paramètrer FM à obtenir, comme l'utilisation d'un algorithme particulier de fusion ou l'interprétation des variables en terme de structure.

6.2 Choix d'implémentation

Pour notre contribution, nous avons décidé d'utiliser FAMILIAR pour soustendre les activités de manipulation de FM. En effet, comme énoncé à la Section 2.3.2, FAMILIAR est le seul outil fonctionnel proposant l'opérateur de fusion, condition sine qua non de notre outil d'extraction de la variabilité. Pour le choix du langage de programmation utilisé, nous avons choisi d'utiliser Java vu que FAMILIAR est codé de cette façon.

Nous avons choisi de rédiger un DSL avec Xtext [63], un framework pour le développement de langages dédiés. Nous considérons que les DSLs sont un bon moyen pour l'utilisateur de se familiariser avec un outil, contrairement à une API dans un langage généraliste.

L'import des données par défaut est le CSV. Ce format a été créé pour représenter des tableaux à deux dimensions. Il n'a donc pas d'autres éléments que des valeurs séparées par des virgules. Son cadre d'utilisation est bien fixé et est d'expressivité pauvre. Il est donc facile de transformer d'autres formats de fichiers semi-structurés en un ensemble de fichiers CSV. De plus, ils sont facilement manipulables, diffusables et sont fortement répandus.

Pour valider notre implémentation, une série de tests unitaires ont été écrits. Nous avons, pour cela, utilisé la librairie Junit [42]. Ils sont décrits à la Section 8.4.

6.3 Sources

Le code source de l'implémentation présenté dans ce chapitre consiste en 5337 lignes de codes réparties en 37 classes. 382 méthodes ont été écrites dont la plupart sont vérifiées par des tests unitaires. Une documentation Javadoc complète a été rédigée et compilée dans le package du projet. Une archive zip est disponible en ligne à l'adresse `http://www.douby.be/dl/realisations/memoire/FmlExtract.zip`. Elle est aussi disponible sur le serveur Subversion `https://www.webkot.be/svn/fmlextract/`. L'utilisateur veillera, pour se connecter, a utiliser comme login « guest » et comme mot de passe « guest ».

Chapitre 7

Etude de cas

Dans ce chapitre, nous expliquons le flux de travail complet pour convertir une SPL existante en un FM à partir d'un exemple. À la Section 7.1, nous décrivons les fichiers de données issus de WikiMatrix, présentés au Chapitre 3. Premièrement, l'utilisateur doit écrire un traducteur entre son formalisme d'encodage des données en entrée et le modèle de conversion décrit à la Section 4.3.1. Cette tâche est présentée à la Section 7.2. Dans un second temps, l'utilisateur paramétrise le processus de conversion en écrivant un ensemble de directives avec le DSL prévu à cet effet (abordé à la Section 7.3). Ce DSL permet, par exemple, de définir l'interprétation de la variabilité en terme de features, de filtrer et d'arranger à ses besoins les données passées en entrée. Après cette étape, l'utilisateur lance la conversion pour récupérer un script FAMILIAR qu'il pourra alors exécuter pour finalement obtenir le FM final. L'étape de lancement du convertisseur est décrite à la Section 7.4. Le script généré est expliqué à la Section 7.5. Ce FM contient l'entièreté de la variabilité des données passées en entrée et filtrées par les directives. Le FM final est décrit à la Section 7.6.

7.1 Données en entrée

Les données de comparaison de moteurs de wiki ont été récupérées sur le site de WikiMatrix[1]. Leur API consiste juste en deux fichiers XML disponibles en ligne. Le premier décrit la structure de données. Nous pouvons voir dans le Listing 7.1 que les balises ***itemgroup*** et ***item*** peuvent être intuitivement traduites dans les concepts de *vues* et de *caractéristiques* du modèle de conversion.

```
1 <itemgroup g_id="1" name="General Features">
2     <item i_id="1" name="Version"/>
3     <item i_id="60" name="Last Release Date"/>
```

1. Le 14 Janvier 2011

```
    <item i_id="2" name="Author"/>
    <item i_id="3" name="URL"/>
    <item i_id="4" name="Free and Open Source"/>
    <item i_id="5" name="License"/>
    ...
</itemgroup>
<itemgroup g_id="18" name="Hosting Features">
    <item i_id="140" name="Storage Quota"/>
    ...
    <item i_id="145" name="Ads allowed"/>
</itemgroup>
```

Listing 7.1 – Extrait du fichier XML de WikiMatrix donnant la structure des données

Le second fichier contient les données dont un extrait est donné par le Listing 7.2. Chaque moteur de wiki décrit est délimité par la balise ***product***. Les balises ***itemgroup*** et ***item*** donnent les données associées aux différentes vues/caractéristiques pour le produit concerné.

```
<product p_id="1" name="DokuWiki">
<general_info name="General Information">

<info name="Description">
DokuWiki is a standards compliant, simple to...</info>
<info name="Record added">2005-11-22</info>
<info name="Last updated">2010-11-08</info>
</general_info>

<itemgroup g_id="1" name="General Features">
    <item i_id="1" name="Version">2010-11-07 Anteater</item>
    <item i_id="60" name="Last Release Date">2010-11-07</item>
    <item i_id="2" name="Author">Andreas Gohr</item>
    <item i_id="3" name="URL">http://www.dokuwiki.org</item>
    <item i_id="4" name="Free and Open Source">Yes</item>
    <item i_id="5" name="License">GPL 2</item>
    <item i_id="51" name="Programming Language">PHP</item>
    ...
</itemgroup>

<itemgroup g_id="18" name="Hosting Features">
    <item i_id="140" name="Storage Quota">Linux, UNI...</item>
    <item i_id="141" name="Bandwidth Quota">No</item>
    ...
</itemgroup>
...
```

Listing 7.2 – Extrait du fichier XML de WikiMatrix donnant les données

L'utilisateur doit maintenant écrire un traducteur entre ces structures XML et le modèle de conversion, structure compréhensible par le convertisseur.

7.2 Création du modèle de conversion

Il y a deux manières de transformer l'entrée XML en un modèle de conversion. La première est de traduire directement les données XML en un objet [2] nommé ConversionModel. Cependant, il peut être plus facile de traduire le formalisme en entrée dans un formalisme plus basique et nativement géré par le convertisseur, c'est-à-dire, CSV. C'est cette deuxième approche que nous allons utiliser dans cette section.

Le format de fichier CSV est un type de donnée semi-structuré. Un fichier CSV consiste en un fichier texte (en opposition aux fichiers dits binaires) qui contient des *valeurs* séparées par des virgules sur chacune des lignes du fichier. En considérant son manque de définition précise, on peut dire que son utilisation est plutôt libre. Nous avons choisi de prendre [56] comme point de départ pour définir nos propres règles constituant un fichier CSV :

- Chaque *rangée* est localisée sur une ligne séparée, délimité par un retour à la ligne (`\n` pour les systèmes UNIX, `\r\n` pour les systèmes Windows).
- Le retour à la ligne de la dernière rangée n'est pas obligatoire.
- La première ligne du fichier est l'*entête* et a le même format qu'une rangée. Chacune des *valeurs* de cette en-tête sont appelées *labels* car elles donnent de l'information à propos des cellules de la colonne associée.
- Dans l'*entête* ou dans une *rangée* il y a un certain nombre de *valeurs* séparées par des virgules. Si une rangée contient moins de *valeurs* que l'en-tête, cette rangée est étendue par des *valeurs* par défaut.
- Tous les espaces et les caractères particuliers sont effacés car ils ne sont pas acceptés comme nom des feature.

Le Listing 7.3 donne un exemple de fichier CSV défini selon les règles énoncées ci-dessus.

```
1 Identifior , License , Language , Storage , LicenseFee
2 PBwiki , Nolimit ,No,No, Yes
3 MoinMoin ,GPL, Python , Files ,No
4 DokuWiki ,GPL2,PHP, Files ,No
5 PmWiki,GPL2,PHP, Files ,No
6 DrupalWiki ,GPL2,PHP, Database , Differentlicences
7 TWiki ,GPL, Perl , FilesRCS , Community
```

2. En faisant référence au langage orienté objet Java, utilisé dans l'implémentation du convertisseur.

```
8 MediaWiki ,GPL,PHP, Database ,No
9 Confluence , Commercial , Java , Database , US10
10 ...
```

Listing 7.3 – Équivalent de la vue *General* de la figure 3.1

Ces règles ne sont pas restrictives par rapport aux formats de fichiers CSV communément définis. Cela implique que pour ainsi dire tous les fichiers CSV vont être acceptés par le module d'entrée du convertisseur. De par sa structure tabulaire, nous pouvons faire le lien direct entre un fichier CSV et une vue : un tableau du modèle de conversion.

Pour traduire le formalisme XML défini à la section précédente en fichiers CSV, l'utilisateur doit créer un fichier CSV (associé à une vue) par ***itemgroup***, une rangée par moteur de wiki concerné et écrire chaque valeur les unes après les autres en les séparant par des virgules. La traduction des fichiers XML du cas WikiMatrix génère 16 fichiers CSV car les données regroupent 16 points de vues de comparaison, ou autant d'***itemgroup***.

Les données étant traduites dans un formalisme compréhensible par notre convertisseur, il faut maintenant que l'utilisateur écrive des directives avec notre DSL pour filtrer et paramétrer notre processus d'extraction.

7.3 Filtrage avec le DSL

Notre approche consiste en la conversion des données de l'utilisateur en un FM unique regroupant l'entièreté de la variabilité. Il est toutefois possible que ce modèle ne corresponde pas tout à fait aux besoins de l'utilisateur. En effet, les 16 fichiers CSV en entrée contiennent beaucoup de données inutiles pour le processus de configuration, comme le numéro de version de l'implémentation du moteur de wiki ou encore le nom et l'adresse mail de son créateur. C'est pourquoi notre approche propose à l'utilisateur de configurer l'extraction en fonction de ses besoins et intérêts. Nous fournissons un DSL pour qu'il puisse paramétriser programmativement le processus d'extraction.

Les directives présentées dans le Listing 7.4 permettent ainsi de filtrer les fichiers CSV crées à l'étape précédente.

```
1 import:
2 // Nous importons seulement les fichiers que nous utilisons (
    parmi les 16 générés à l'étape précédente)
3 "CSV/Output.csv" as output
4 "CSV/SystemRequirements.csv" as sysrequirements
5 "CSV/GeneralFeatures.csv" as generalFts
```

```
6 "CSV/MediaandFiles.csv" as media
7 "CSV/SpecialFeatures.csv" as special
8
9 name: "WikiMatrix"
10 // Nom de la feature racine du FM final
11 structure:
12   output moveIn special
13   // Place toutes les données de la vue output dans la vue
       special
14   media moveIn special
15
16 default:
17 // Directives portant à toutes les vues
18   parsing:
19       key: "Identifier"
20   only_products: "MoinMoin" "DokuWiki" "MediaWiki" "PmWiki" "
       TWiki" "DrupalWiki" "Confluence" "PBwiki"
21   // Filtre les entités pour garder uniquement "MoinMoin" "
       DokuWiki" ...
22
23 view: generalFts
24   rootname: "General"
25   // Nom effectif de la vue generalFts
26   only_features: "License" "Language" "Storage" "LicenseFee"
27
28 view: special
29   rootname: "Functionalities"
30   only_features: "Unicode"
31
32 view: output
33   only_features: "RSS"
34
35 view: media
36   only_features: "Flash" "ImageEditing"
37
38 view: sysrequirements
39   rootname: "Requirements"
40   except_features: "Webserver"
41   // Filtre la caractéristique "Webserver"
```

Listing 7.4 – Directives de filtrage pour obtenir les 3 Figures 3.1, 3.2 et 3.3

L'ensemble des directives du Listing 7.4, permet de restreindre les données à 3 vues et à 8 entités. Elles sont représentées visuellement par les Figures 3.1, 3.2 et 3.3.

Maintenant que les directives sont écrites, il faut que l'utilisateur lance le processus d'extraction.

7.4 Lancement du convertisseur

Pour créer un modèle de conversion à partir de fichiers CSV et des directives écrites à l'étape précédente, il faut utiliser l'objet StoFMLReader de notre librairie Java. Cette opération consiste en quelques lignes Java reprises par le Listing 7.5.

```
1  package useCases.wikiMatrixDSL;
2  import CSVtoFML.FMLConvert;
3  import CSVtoFML.StoFMLReader;
4
5  public class Launcher {
6    public static void main(String[] args) {
7      String path = "wikiMatrixDSL/filters/";
8     // Dossier contenant le fichier de directive
9     StoFMLReader stoFMR = new StoFMLReader();
10    // Instanciation de lecteur de directives
11     try {
12       FMLConvert converter =
13      stoFMR.execute(path + "filterSmall.stofm");
14       // Début de la conversion à partir du fichier de
             directives "filterSmall.stofm"
15     } catch (Exception e) {
16       e.printStackTrace();
17       System.exit(1);
18     }
19   }
20 }
```

Listing 7.5 – Lancement du convertisseur à partir d'un fichier de directives

Maintenant que le modèle de conversion est établi, l'utilisateur doit récupérer le script FAMILIAR intermédiaire qui, une fois exécuté, produira le FM final.

7.5 Script généré

Pour récupérer le script FAMILIAR qui permet de générer le FM final, il faut appeler la méthode ***getFinalFAMILIARScript()*** de l'objet déclaré à la section précédente. Le code Java présenté au Listing 7.6 permet d'imprimer ce script au flux standard de sortie.

```
1 System.out.println(
2       converter.getFinalFAMILIARScript());
```

Listing 7.6 – Récupérer le script FAMILIAR

La Figure 7.1 reprend un ensemble de métriques pour rendre compte du FM généré. Les colonnes renseignent du nombre de données à l'origine, le nombre d'éléments filtrés, le nombre gardé pour la fusion et donne certaines valeurs relatives au FM généré. Les lignes, quant à elles, renseignent le nombre de fichiers (ou vues), de caractéristiques, de rangées, d'entités et de valeurs textuelles qui ont été remplacées par une expression régulière. Quelques indications sur le FM final sont données, comme le nombre de features différentes (ayant un nom différent), de valeurs induisant une structure *Obligatoire*, *Optionnelle*, feature morte et textuelles.

	Origine	**Filtré**	**Gardé**	**FM Final**
Fichiers	16	13	3	-
Caractéristiques	147	19	11	-
Rangées	2016	?	24	-
Entités	126	118	8	-
Valeurs	18522	18434	88	-
Valeurs remplacées	-	29	-	-
Features différentes	-	-	-	41
Valeurs Obligatoires	-	-	-	33
Valeurs Optionnelles	-	-	-	2
Dead Features	-	-	-	25
Valeurs Textuelles	-	-	-	28

FIGURE 7.1 – Point de vue *General*

Le Listing 7.7 donne un extrait du script FAMILIAR généré par l'appel du Listing 7.6.

```
1 // FMs d'étapes 1 pour l'entité MoinMoin
2 MoinMoin_Requirements = FM (Requirements : [OperatingSystem] [
    RootAccess] [OtherRequirements] ; Requirements -> !
    OperatingSystem; Requirements -> !RootAccess; Requirements ->
     !OtherRequirements; )
3 MoinMoin_General = FM (General : License ProgrammingLanguage
    DataStorage [LicenseCost_Fee] ; License : GPL ;
    ProgrammingLanguage : Python ; DataStorage : Files ; General
    -> !LicenseCost_Fee; )
4 MoinMoin_Functionalities = FM (Functionalities : UnicodeSupport
     RSSFeeds EmbeddedFlash [ImageEditing] ; Functionalities -> !
    ImageEditing; )
5
6 // FM d'étape 2 pour l'entité MoinMoin
7 fm_MoinMoin = FM(WikiCompare : ID ; ID : MoinMoin ; )
8 foreach(f in MoinMoin_*) do
9 insert f into fm_MoinMoin.WikiCompare with mand
10 end
11 // Optimisation. Retire les FMs d'étape 1 de la mémoire.
```

```
12 removeVariable MoinMoin_Requirements
13 removeVariable MoinMoin_General
14 removeVariable MoinMoin_Functionalities
15
16 // FMs pour l'entité PBwiki
17 PBwiki_Functionalities = FM (Functionalities : [UnicodeSupport]
       RSSFeeds EmbeddedFlash ImageEditing ; EmbeddedFlash :
      Indentedblock ; ImageEditing : Number1Number2 ;
      Functionalities -> !UnicodeSupport; )
18 PBwiki_Requirements = FM (Requirements : [OperatingSystem]
      RootAccess [OtherRequirements] ; Requirements -> !
      OperatingSystem; )
19 PBwiki_General = FM (General : License [ProgrammingLanguage] [
      DataStorage] LicenseCost_Fee ; License : Nolimit ; General ->
       !ProgrammingLanguage; General -> !DataStorage; )
20 fm_PBwiki = FM(WikiCompare : ID ; ID : PBwiki ; )
21 foreach(f in PBwiki_*) do
22 insert f into fm_PBwiki.WikiCompare with mand
23 end
24 removeVariable PBwiki_Functionalities
25 removeVariable PBwiki_Requirements
26 removeVariable PBwiki_General
27
28 // Retiré les FMs des autres entités par souci de lisibilité
29
30 // Etape 3, union stricte de tous les FMs d'étape 2
31 finalFM = merge sunion fm_*
```

Listing 7.7 – Script FAMILIAR généré à partir des directives du Listing 7.4

Par souci de lisibilité, les scripts intermédiaires pour les FMs *DokuWiki*, *PmWiki*, *DrupalWiki*, *TWiki*, *MediaWiki* et *Confluence* ont été retirés. Les lignes 2, 3 et 4 correspondent aux trois FMs générés par l'étape 1 du processus d'extraction pour le moteur de wiki *MoinMoin*, c'est-à-dire qu'elles contiennent la variabilité associée aux différentes vues pour une entité particulière. Les lignes 7 à 10 donnent le script généré par l'étape 2 de notre convertisseur. Les lignes 12 à 14 sont présentes par souci d'optimisation, comme énoncé dans la Section 8.3.2. Elles permettent de décharger de la mémoire des FMs plus utilisés. Les lignes 17 à 26 sont équivalentes aux lignes précédemment énoncées, mais dédiées à l'entité *PBwiki*. Pour finir, la ligne 31 opère une fusion sur l'ensemble des FMs d'étape 2.

Pour récupérer le FM final, l'utilisateur doit exécuter le script FAMILIAR obtenu à cette étape.

7.6 Génération du FM final

Pour récupérer le FM final, il faut appeler la méthode ***getFinalFeatureModel(boolean)*** de l'instance ***converter***, précédemment déclarée. Le

paramètre booléen permet d'obtenir une génération verbeuse, retraçant les différentes étapes de conversion.

```
1       System.out.println(
2      converter.getFinalFeatureModel(false));
```

Listing 7.8 – Récupération du FM final

Le Listing 7.9 donne le FM final généré par notre processus d'extraction. Il est obtenu en appelant la méthode donnée dans le Listing 7.8. Par souci de lisibilité, nous avons retiré un certain nombre de contraintes. Il faut noter que celles-ci sont assez complexes et ne sont pas facilement compréhensibles ni modélisables manuellement. La ligne 1 stipule que le FM est placé dans la variable ***FinalFM***. La ligne 4 donne la première production, c'est-à-dire qu'elle contient la racine, les différentes vues et la feature ID. La ligne 7 donne le développement de la vue *Functionnalities* qui contient deux groupes : un *groupe Ou* et un *groupe Et*. La ligne 12 nous montre l'ensemble des identifiants dans un *groupe Alternatif*. Les contraintes sont renseignées à partir de la ligne 24.

```
1  FinalFM = FM(
2  // WikiCompare est la feature racine
3  // Functionalities , Requirements et General sont les 3 vues
4  WikiCompare: Functionalities Requirements General ID ;
5
6  // Association des caractéristiques aux 3 vues
7  Functionalities: (UnicodeSupport|ImageEditing)+ EmbeddedFlash
       RSSFeeds ;
8  Requirements: (OperatingSystem|OtherRequirements)? [RootAccess]
       ;
9  General: (LicenseCost_Fee|DataStorage)+ License [
       ProgrammingLanguage] ;
10
11 // Liste des noms d'entités sous la feature spéciale ID
12 ID: (MoinMoin|TWiki|PmWiki|PBwiki|DrupalWiki|Confluence|DokuWiki
       |MediaWiki) ;
13
14 // Liste des caractéristiques et leurs sous-features
15 ImageEditing: [Number1Number2] ;
16 EmbeddedFlash: [Indentedblock] ;
17 OtherRequirements: [Java5] ;
18 License: (GPL|Commercial|GPL2|Nolimit) ;
19 LicenseCost_Fee: (Differentlicences|US10|Community)? ;
20 ProgrammingLanguage: (Java|Python|PHP|Perl) ;
21 DataStorage: (Files|FilesRCS|Database) ;
22
23 // Quelques contraintes
24 (Confluence -> Java);
25 (Number1Number2 -> Indentedblock);
```

```
26 (FilesRCS -> TWiki);
27 (TWiki -> Perl);
28 (Files -> UnicodeSupport);
29 (Java -> Confluence);
30 ((((((((((((((((((((((((true & !FilesRCS) & !Java5) & !Database
       ) & !Nolimit) & !OtherRequirements) & !Differentlicences) & !
       Number1Number2) & !Perl) & !Java) & !PBwiki) & !DrupalWiki) &
        !Python) & !OperatingSystem) & !Community) & !RootAccess) &
       !Commercial) & !TWiki) & !MoinMoin) & !GPL) & !US10) & !
       PmWiki) & !MediaWiki) & !Indentedblock) & !Confluence) ->
       !((((((((((((((((true & GPL2) & PHP) & true) &
       LicenseCost_Fee) & true) & true) & true) & Files) &
       ProgrammingLanguage) & UnicodeSupport) & ImageEditing) &
       DokuWiki) & DataStorage) & true) & true) & true));
31 ((((((((((((((((((((((((true & !FilesRCS) & !Java5) & !Database)
        & !Nolimit) & !GPL2) & !OtherRequirements) & !
       Differentlicences) & !Number1Number2) & !Perl) & !Java) & !
       PBwiki) & !DrupalWiki) & !OperatingSystem) & !Community) & !
       Commercial) & !TWiki) & !DokuWiki) & !US10) & !PmWiki) & !PHP
       ) & !MediaWiki) & !Indentedblock) & !Confluence) ->
       !((((((((((((((((MoinMoin & GPL) & true) & true) &
       LicenseCost_Fee) & true) & true) & Files) & true) &
       ProgrammingLanguage) & Python) & UnicodeSupport) &
       ImageEditing) & RootAccess) & DataStorage) & true) & true) &
       true));
32 (Commercial -> Confluence);
33
34 // Les autres contraintes ont été retirées par soucis de
       lisibilité
```

Listing 7.9 – Extrait du FM final résultant de l'extraction

Chapitre 8

Évaluation

Dans ce chapitre, nous évaluons le processus d'extraction décrit au Chapitre 4. Pour cela, nous parlons de la performance et de la montée en charge de notre approche. En effet, elle se base sur des algorithmes comme, par exemple, la fusion, qui est elle-même basée sur un problème SAT NP-complet. Cette évaluation est primordiale dans le cadre de ce mémoire. Elle permet de rendre compte l'utilisabilité et des limites réelles de notre convertisseur qui, dans un cas échéant, pourrait être utilisé dans l'industrie.

Dans la Section 8.1 nous décrivons les tests que nous exécutons pour déterminer les limites de scalabilité de notre contribution. Nous y déterminons le mode opératoire, l'environnement d'exécution, les différents ensembles de données en entrée ainsi que différentes métriques permettant de mesurer la performance. Ensuite, nous exposons deux scénarios. Le premier, à la Section 8.2, donne un cas d'utilisation classique en utilisant les paramètres par défaut de FAMILIAR. Le second, à la Section 8.3, décrit des optimisations pour une meilleure montée en charge. Ces deux sections présentent les résultats des tests. Finalement, dans la Section 8.4, nous discutons de la validité des modèles générés.

8.1 Établissement des tests

Pour identifier les limites de scalabilité de notre approche, nous mettons en œuvre une série de tests. Ils se veulent synthétiques dans la façon de créer les modèles de conversion, mais ils sont basés sur les données fournies par WikiMatrix (introduit au Chapitre 3) qui constitue un cas d'utilisation réaliste. Si les résultats montrent la montée en charge de l'application pour un grand nombre de features, nous pourrons conclure que ceux-ci sont généralisables à d'autres cas, ainsi qu'à l'industrie.

Nous avons créé un générateur de directives de filtrage qui crée des mo-

dèles de conversions à partir de 3 paramètres :

- ***nVues***, le nombre de vues ;
- ***nCaracteristiques***, le nombre de caractéristiques ;
- ***nEntites***, le nombre d'entités / produits.

Au niveau de la scalabilité, ces paramètres ne sont utiles que pour générer des données en entrée plus ou moins importantes. L'extraction de la variabilité se fait sur un ensemble de valeurs équivalentes à $nValeurs = nVues * nCaracteristiques * nEntites$. Nous avons décidé de ne pas considérer comme important les paramètres séparément, car nous estimons que ceux-ci n'influencent que la structure du FM final, pas le temps de calcul.

Les modèles générés contiennent des valeurs prises aléatoirement à partir des données de WikiMatrix. Grâce à notre générateur, nous lançons les tests en créant à la volée des modèles de conversions contenant progressivement de plus en plus de valeurs. A chaque exécution, nous générons un fichier rapportant différentes métriques du FM généré. Elles visent à caractériser plus finement les modèles de conversion associés. Voici les plus importantes :

- ***nObligatoire***, le nombre de valeurs *Obligatoires* ;
- ***nOptionnelles***, le nombre de valeurs *Optionnelles* ;
- ***nDeadFeatures***, le nombre de valeurs features mortes ;
- ***nTextualValues***, le nombre de valeurs textuelles ;
- ***tempsExec***, le temps d'exécution du processus d'extraction pour générer le FM final. Celui-ci est mesuré en utilisant des timestamps. $t\_fin - t\_debut = t\_exec$ avec t_fin et t_debut respectivement les timestamps de fin et de début de l'exécution de l'algorithme et t_exec le temps d'exécution en secondes ;
- ***nUniqueFeature***, le nombre de features dans le FM final. Nous désignons par « feature unique » une valeur produisant une nouvelle feature dans le FM final.

La configuration de test est un ordinateur de bureau classique. Il est composé d'un processeur Intel Core 2 duo E6750, 6Go Ram à 800Mhz de fsb et un disque dur Western Digital Raptor 150 Go 15000tr/min. Le tout tourne sur Ubuntu 11.04 desktop 64bit, OpenJDK 20.0-b11, eclipse 3.6.2 et une version de FAMILIAR compilée avec les sources datant du 22 Mai 2011.

Grâce aux métriques présentées dans cette section, nous allons déterminer la montée en charge brute de notre approche et vérifier si les interprétations en terme de structure des valeurs particulières influent sur la performance du processus d'extraction.

8.2 Scénario de base

Nous allons tester dans cette section la scalabilité du processus d'extraction avec notre outil et FAMILIAR paramétré par défaut. En lisant la littérature [25], nous avons remarqué la limite pratique de certains algorithmes utilisés par l'opérateur de fusion de FAMILIAR, c'est pourquoi, nous le présentons en détail, en vue de les identifier, à la Section 8.2.1. Finalement, les résultats de l'application des tests et l'interprétation de ceux-ci sont donnés à la Section 8.2.2.

8.2.1 Algorithme de fusion de FAMILIAR

Comme décrit dans la Section 2.2.4, l'algorithme de fusion est utilisé pour créer une représentation compacte de tous les FMs d'entités. Comme illustré par la Figure 8.1, il encode les FMs en entrée, ***FM1*** à ***FMx*** (avec ***x*** le nombre de FMs), en des formules propositionnelles, ***FP1*** à ***FPx***. Il effectue alors certaines opérations booléennes sur celles-ci, en fonction du mode de fusion choisi, ce qui produit une nouvelle formule propositionnelle nommée ***FPFusion***. Cette dernière est alors passée dans un algorithme de reconstruction de hiérarchie [25]. L'algorithme peut restaurer la hiérarchie résultant de la fusion des FMs donnés en entrée en utilisant prioritairement des contraintes implicites : les liens parent-enfants (différents groupes et relations).

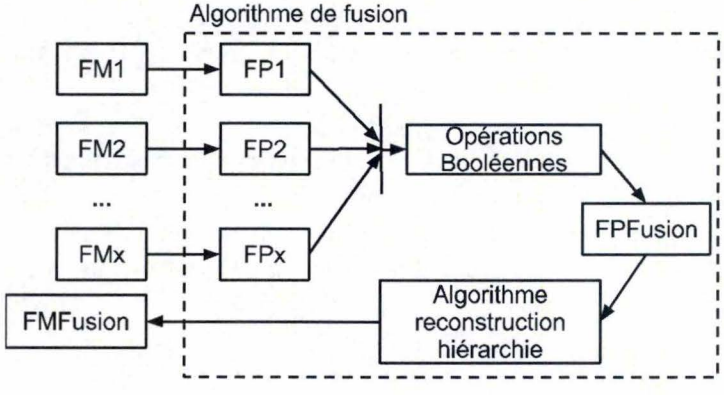

FIGURE 8.1 – Algorithme de fusion classique

8.2.2 Résultats et interprétation

Selon le cadre décrit à la Section 8.1, nous avons lancé un test comportant 20 entités, 12 vues avec 5 caractéristiques chacune. Le résultat du test est illustré par la Figure 8.2. Avec ces paramètres, 1200 valeurs sont prises

en considération et le FM final contient 391 features. Nous avons lancé le script FAMILIAR généré par notre processus d'extraction. Nous avons décidé de l'arrêter après 10 minutes d'exécution. Malgré que le test porte sur des données de petite taille, le temps d'exécution est important.

Test	Valeurs	Secondes	Features	Obligatoire	Dead	Optionnel	Textuelle
20_12_5	1200	>600	391	286	280	281	353

FIGURE 8.2 – Résultat d'évaluation du scénario de base

Nous constatons que la performance est un problème majeur dans notre approche. Nous avons identifié certains goulots d'étranglement, comme par exemple, l'algorithme de Czarnecki [1]. C'est pourquoi nous avons implémenté différentes optimisations et conduit une batterie de tests afin de vérifier la scalabilité de notre approche optimisée. Nous les décrivons dans la section suivante.

8.3 Scénario optimisé

Dans cette section, nous présentons les différentes techniques d'optimisation du processus d'extraction, ainsi que de FAMILIAR. Celles-ci sont utilisées à plusieurs niveaux, aussi bien dans la création d'un processus de fusion dédié à notre approche, décrit à la Section 8.3.1, que dans l'utilisation de librairies efficaces et d'astuces algorithmiques, exposées à la Section 8.3.2. Elles ont été développées en vue de répondre au problème de montée en charge énoncé dans la section précédente. Ensuite, dans la Section 8.3.3, nous évaluerons notre approche par une série de tests et donnerons l'interprétation des résultats.

8.3.1 Fusion par structure approximée

L'algorithme de fusion que nous avons développé particulièrement pour notre processus d'extraction est décrit par la Figure 8.3. Nous l'avons appelé *fusion par structure approximée* car nous utilisons une approximation, un squelette du FM à obtenir. Il est construit en inférant sa hiérarchie à partie des données en entrée de notre processus d'extraction ainsi que des directives renseignées par l'utilisateur. La fusion s'effectue selon les étapes suivantes : (1) les FMs en entrée, ***FM1*** à ***FMx***, sont encodés en des formules propositionnelles, nommées ***FP1*** à ***FPx*** ; (2) certaines opérations booléennes sont exécutées selon le mode de fusion de base et on obtient ***FPFusion***, une expression booléenne résultant de la fusion ; (3) on encode la structure

1. En effet, cet algorithme, étant de complexité élevée pour la construction des *groupements Ou*, monte difficilement en charge.

approximée en une formule propositionnelle, nommée ***FPApproximé*** ; (4) on exécute la fusion en *mode différence* sur les deux formules ***FPFusion*** et ***FPApproximé***. On obtient alors une formule booléenne qui correspond à un ensemble de contraintes ; (5) on ajoute les contraintes obtenues à la structure approximée, qui est une déjà hiérarchie de features. On obtient alors, un FM qui a le même ensemble de configurations que celui résultant de l'algorithme de fusion classique utilisant l'algorithme de reconstruction de hiérarchie [25], à la différence près que l'opérateur utilisé est de faible complexité. Les opérations qui sont en bleu à la Figure 8.3 sont celles qui remplacent l'algorithme de reconstruction de hiérarchie.

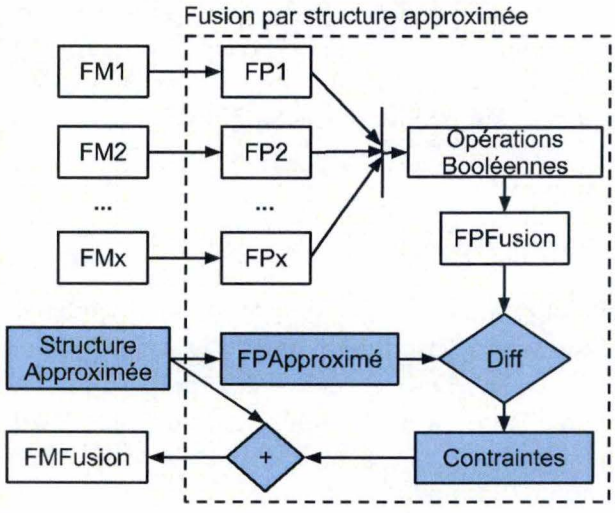

FIGURE 8.3 – Fusion par structure approximée

La clé de notre solution est la création de la structure approximée. L'idée est de créer une généralisation (nous la nommerons par la suite ***FMA***) du FM à obtenir (***FMF*** ou FM de Fusion). Pour ce faire, nous utilisons un processus qui parcourt les différents tableaux du modèle de conversion et les données associées. L'algorithme est de complexité linéaire et est décrit comme suit :

- Créer un nouveau FM désigné ***FMA***.
- Nommer la racine de ***FMA*** par le rootFeatureName du modèle de conversion.
- Créer la feature particulière ID sous la racine de ***FMA***.
- Ajouter sous ID l'ensemble des identifiants des entités dans un *groupe Alternatif*. Celui-ci est obtenu en faisait l'union des identifiants de chacune des vues.

- Itération pour chaque vue :
  - Créer une feature *Obligatoire* avec le nom de la vue et la placer sous la racine. Nous l'identifions par ***FVue***.
  - Itération pour chaque caractéristique de ***FVue*** :
    - Créer une feature *Obligatoire* avec le nom de la caractéristique et la placer sous ***FVue***. Nous l'identifions par ***FCar***.
    - Itérer pour chaque valeur associée à la caractéristique pour inférer la relation entre ***FVue*** et ***FCar*** :
      - Si la valeur correspond à l'interprétation par la structure *Obligatoire* (par exemple, « Yes »), ignorer (car la relation actuelle entre ***FVue*** et ***FCar*** est déjà *Obligatoire*).
      - Si, la valeur correspond à une structure *Optionnelle*, transformer la relation entre ***FVue*** et ***FCar*** en une relation *Optionnelle*.
      - Si la valeur correspond à une structure *dead feature*, transformer la relation entre ***FVue*** et ***FCar*** en une relation *Optionnelle*
      - Si la valeur est textuelle, ajouter une sous-feature *Obligatoire* à ***FCar***.
    - Si toutes les valeurs associées à ***FCar*** induisent des *dead features*, ajouter une contrainte impliquant la négation de ***FCar***.
    - Si des valeurs textuelles sont présentes pour la caractéristique :
      - S'il n'y a qu'une valeur textuelle et que ***FCar*** est *Optionnelle*, changer la relation entre ***FCar*** et la valeur en *Optionnelle*.
      - S'il y a plusieurs valeurs textuelles et que ***FCar*** est *Optionnelle*, placer ces valeurs dans un *groupe Mutex*.
      - S'il y a plusieurs valeurs textuelles et que ***FCar*** est *Obligatoire*, placer ces valeurs dans un *groupe Alternatif*.
  - Si chaque caractéristique de ***FVue*** est *Optionnelle*, modifier la relation entre la racine et ***FVue*** en une relation *Optionnelle*.
- Transformer cette hiérarchie dans le langage de modélisation de FAMILIAR.

Il est intéressant de noter que pour obtenir une caractéristique *Obligatoire*, il est nécessaire que toutes les valeurs associées soient des structures *Obligatoires*. À l'inverse, pour obtenir une vue *Optionnelle*, il est nécessaire que toutes les caractéristiques associées aient des relations *Optionnelles*. Notre approximation ***FMA*** est uniquement composé des *groupes Et*, *Alternatif* et *Mutex*. En effet, le *groupe Ou* est trop difficile à inférer avec une analyse du modèle de conversion (stratégie syntaxique).

8.3.2 Optimisations diverses

Le script FAMILIAR, qui résulte de l'extraction de variabilité, comporte un grand nombre de FMs. Tous les charger en mémoire prend beaucoup de place pour un catalogue de produit un minimum conséquent. C'est pourquoi nous avons identifié 3 optimisations possibles, liées aux chargements dynamiques

de ceux-ci.

1. Comme expliqué dans la Section 4.3.3, le processus d'extraction se déroule en 3 étapes. La première consiste en la génération d'un FM par entité et par vue. Ils sont alors chargés en mémoire. La seconde, crée des FMs qui résultent de l'agrégation des FMs d'étape 1. À partir du moment où un FM d'étape 1 est inséré dans un FM d'étape 2, il n'est plus utilisé. C'est pourquoi, nous insérons dans le script FAMILIAR généré une opération de suppression des FMs concernés après chaque nouveau FM d'étape 2. Il suffit d'appeler « removeVariable nomDeFM » pour retirer***nomDeFM*** de la mémoire.
2. Nous avons modifié FAMILIAR pour que, lors du chargement d'un FM, son expression booléenne soit calculée et enregistrée dans une variable en mémoire. Cette optimisation permet de ne pas la recalculer à chaque appel.
3. Nous avons changé la structure d'appels de FAMILIAR pour que celui-ci charge un FM en mémoire uniquement quand celui-ci est appelé.

Ces optimisations sont mineures en termes de code et d'impact, mais elles peuvent, dans certains cas, amener des gains de performances non négligeables. Leur utilisation n'importe évidemment pas un changement dans l'ordre de grandeur des opérations exécutées.

En lisant la littérature [48], nous avons identifié une heuristique pour la compilation de BDD qui permet de mieux monter en charge (l'algorithme de BDD de base plafonne à 2000 features différentes). Nous avons donc décidé d'intégrer S.P.L.A.R. pour cette tâche.

8.3.3 Résultats et interprétation

Nous avons effectué une batterie de tests, selon le cadre décrit à la Section 8.1, en faisant varier les paramètres ***nVues***, ***nCaracteristiques*** et ***nEntites***. Nous avons repris les différents résultats bruts dans le tableau de la Figure 8.4. Les différents tests sont nommés, par convention, selon leurs paramètres : ***nEntites_nVues_nCaracteristiques***. L'ensemble de nos tests sont scalables, du moins, jusqu'au test ***120_13_30*** inclus. Nous déterminons comme non-scalable un test dont la complexité en temps ou en mémoire explose de manière exponentielle. Nous n'avons pas jugé intéressant d'étendre les tests au-delà. Premièrement, car l'entièreté de la variabilité de WikiMatrix est convertie (sauf 3 vues contenant des informations inutiles pour le processus de configuration, par exemple, le nom des auteurs des différents moteurs de wiki ainsi que le numéro de version actuelle du programme). Deuxièmement, car le temps d'exécution de l'algorithme devient

trop important pour une utilisation pratique. En effet, le dernier test comportant 13680 valeurs a pris presque 24 minutes pour créer le FM compact. À partir de données en entrée d'une certaine taille, l'utilisateur veillera donc à inclure le processus d'extraction dans un batch. Troisièmement, car nous estimons que travailler sur un FM de cette taille lors de la tâche de configuration devient une tâche difficile pour l'utilisateur. L'analyse du tableau à la Figure 8.4 a permis de soulever certaines remarques à propos de la scalabilité de notre approche. Nous les exposons dans la suite de cette section.

Test	Valeurs	Secondes	Features	Obligatoire	Dead	Optionnel	Textuelle
30_5_5	750	12	262	223	178	104	245
20_11_5	1100	14	351	277	257	248	318
20_12_5	1200	15	391	286	280	281	353
20_10_12	1800	21	526	439	474	364	523
20_11_12	1920	21	550	473	515	384	548
20_11_15	2000	22	570	494	543	395	568
40_11_12	3840	61	899	1141	1007	527	1165
40_11_15	4000	63	928	1192	1064	540	1204
80_11_12	7680	317	1542	2093	2103	1097	2387
120_10_8	9000	671	1732	2441	2415	1296	2848
120_11_15	12000	1000	2126	3314	3353	1701	3632
120_13_30	13680	1433	2532	3560	3911	2056	4153

FIGURE 8.4 – Résultat d'évaluation du scénario optimisé

Considérons le graphique de la Figure 8.5. Il montre l'évolution du temps d'exécution selon le nombre de features uniques. Nous pouvons remarquer que le temps d'exécution augmente en 3 étapes. La première concerne les tests ***30_5_5*** au ***20_11_15***. Il est dans le même ordre de grandeur, c'est-à-dire entre 12 et 22 secondes pour un écart de 308 features. La deuxième consiste en une étape intermédiaire qui regroupe les tests ***40_11_12*** et ***40_11_15*** ayant respectivement 899 et 928 features. Le temps d'exécution de ***40_11_15*** est pour ainsi dire triple de ***20_11_15*** alors que le nombre de features ne fait que d'augmenter de 63%. Nous constatons donc un ralentissement dans la performance par rapport à la première série de tests. La troisième étape concerne les tests ***80_10_8*** à ***120_13_30***. Elle amène une cassure nette en terme de performance entre le test ***40_11_15*** et ***80_10_8***. En effet, il y a un bond de 63 à 317 secondes pour une augmentation de 928 à 1542 features. C'est-à-dire que pour une augmentation de plus de 5 fois le temps d'exécution, il y a une augmentation de 66% de features. Cette tendance se confirme pour les tests jusqu'à ***120_13_30***. Nous supposons donc que lorsque notre algorithme dépasse la barre des 1000 features, le temps d'exécution devient gênant pour l'utilisateur. En regardant les résultats, nous supposons aussi que la scalabilité de notre approche dépend uniquement du nombre de features différentes dans le FM final.

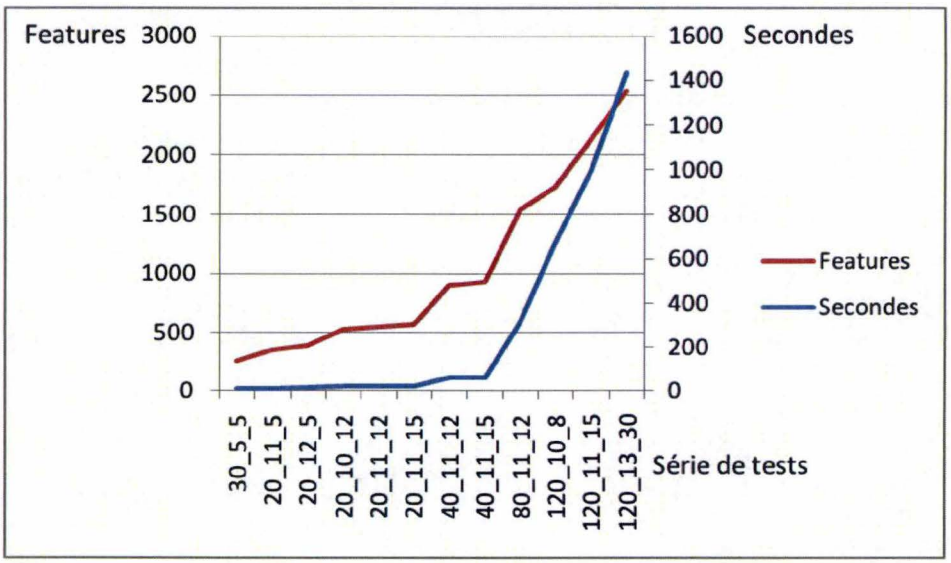

FIGURE 8.5 – Évolution du nombre de features différentes et le temps d'exécution

À la vue de la constatation précédente, nous pourrions penser que l'accroissement soudain du temps d'exécution au seuil des 1000 features uniques est du à la nature des variables considérées. Selon la Figure 8.6, qui montre l'évolution des proportions de différents types de valeurs, il n'en est rien. En effet, les différentes proportions varient entre chacun des tests, mais pas de manière suffisante pour expliquer une telle envolée du temps d'exécution. Ce graphique infirme donc que le nombre de features de types différents impacte les performance de l'extraction. Cette constatation est quand même à nuancer. En effet, si l'utilisateur ajoute des valeurs qui ne créent pas de features uniques, il augmentera toutefois le temps d'exécution vu que l'algorithme devra traiter les nouvelles valeurs.

La Figure 8.7 nous montre que plus il y a de valeurs considérées, moins il y a proportionnellement de features différentes. On peu imaginer que plus il y a de valeurs, plus la probabilité d'ajouter une nouvelle feature au FM final diminue.

8.4 Validité du FM généré

La validité du FM produit par le processus d'extraction est difficile à définir. En effet, la génération de ce FM est associée à un certain nombre d'interprétations du catalogue de produits. Nous dirons que le FM extrait

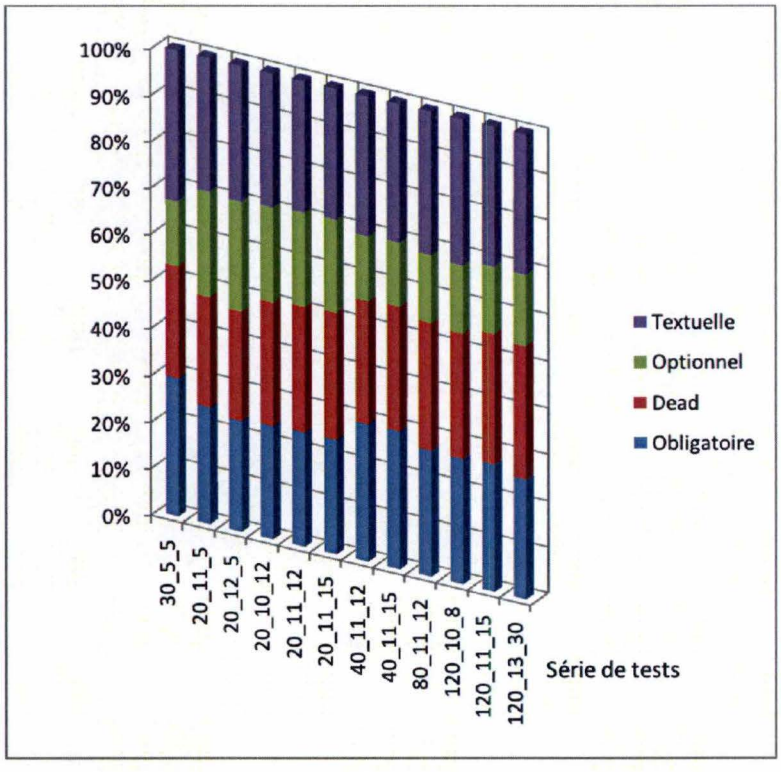

FIGURE 8.6 – Évolution des proportions des différents types de valeurs

est valide si l'ensemble des configurations de celui-ci correspond à l'ensemble des produits du catalogue augmenté des configurations résultant de l'interprétation des valeurs. De plus, pour être correct, notre convertisseur doit fournir à l'utilisateur des fonctionnalités permettant d'exprimer ses besoins et de les appliquer sans effets indésirables ou bugs.

Des tests unitaires ont été écrits pour valider le FM à deux niveaux :

- Vérifier si le FM final est correct. Nous testons toutes les fonctions de toutes les étapes du processus d'extraction, comme l'import des fichiers CSV, les directives associées au DSL, les features et relations des FMs intermédiaires ainsi que la structure et configuration du FM généré. Les tests sont basés sur le modèle : (1) donner une entrée visant à tester une limite particulière de l'algorithme ; (2) comparer la sortie de l'algorithme avec le résultat à obtenir créé manuellement.
- Nous vérifions si les constructions approximées de notre approche optimisée sont correctes avec une stratégie syntaxique, c'est-à-dire que le squelette en sortie de notre algorithme est équivalent à un squelette

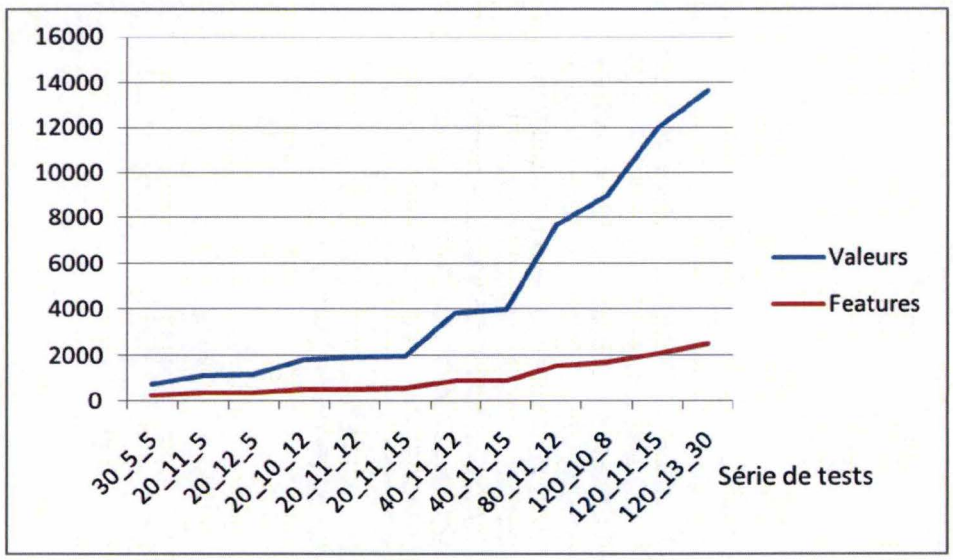

FIGURE 8.7 – Évolution du nombre de features différentes par rapport aux valeurs

créé manuellement. Pour compléter notre test, nous vérifions, en plus, à partir de différents ensembles de données, si les squelettes de FM sont bien une généralisation de la formule propositionnelle du FM résultant.

L'entièreté des tests unitaires passe avec succès. Il peut toutefois subsister des bugs non couverts par nos tests.

Il peut être intéressant de compléter la définition de la validité du FM final, énoncée ci-dessus, par le concept d'expressivité. En effet, comme expliqué à la Section 2.1, les contraintes implicites, c'est-à-dire les relations de groupe, renforcent la sémantique de la hiérarchie de features. Elles évitent d'utiliser des contraintes booléennes, dites explicites, qui sont peu lisibles par l'utilisateur de par leur complexité. Un FM bien modélisé est un FM contenant le plus d'informations sémantiques possible. Comme soulevé dans la Section 8.3.1, notre optimisation par approximation de structure n'utilise pas les *groupes Ou*. Ils sont par conséquent remplacés par des contraintes explicites qui peuvent être conséquentes. Il y a donc, à ce niveau, une perte de qualité dans la génération de la structure du FM final par rapport à l'utilisation de l'algorithme de reconstruction de hiérarchie [25]. Cependant, nous trouvons que l'optimisation utilisée est un bon compromis. Elle apporte une faible perte de lisibilité pour une grande augmentation de la scalabilité.

Chapitre 9

Related Work

Nous présentons dans ce chapitre l'ensemble des travaux connexes aux différentes contributions de ce mémoire. Nous avons parcouru la littérature afin de trouver des publications intéressantes au sujet : (1) de l'extraction de la variabilité de catalogues de produits existants ou de sources de données semi-structurées ; (2) du processus de veille technologique des travaux dont le nôtre dépend ; (3) des travaux dans d'autres domaines comportant des idées intéressantes. Ils sont abordés respectivement aux Sections 9.1, 9.2 et 9.3.

9.1 Extraction de la variabilité

Comme expliqué dans la Section 2.3.2, notre approche est sous-tendue par FAMILIAR, ce qui amène intrinsèquement un lot d'avantages et d'inconvénients. En effet, nous avons défini dans le Chapitre 2 qu'un « bon » FM est un FM expressif. Pour cela, nous essayons d'utiliser le plus de contraintes implicites possible. Cependant, nous sommes limités par le formalisme que nous utilisons pour représenter le FM. S'il est possible de créer de façon scalable des FMs avec des *groupes Ou* et encore mieux avec des *groupes à Cardinalité*, l'utilisateur sera d'autant plus satisfait. De la même manière, certains catalogues de produits utilisent des intervalles de valeurs numériques. Le type entier ou réel peut être très utile pour, par exemple, donner un prix à un produit. Ce sont quelques-unes des extensions de notre approche qui permettraient de générer de meilleurs FMs. Il peut aussi être intéressant d'étudier le concept d'attribut [11, 10] et en quoi celui-ci pourrait améliorer la qualité du FM généré par notre approche.

Dans [12], il est présenté un benchmark sur la variabilité liée aux configurations avec Kconfig du noyau Linux. Il est utile de remarquer que Kconfig utilise uniquement des contraintes binaires dont la consistance est vérifiée par un solveur 2-SAT de complexité polynomiale contrairement à FAMILIAR qui utilise des clauses ternaires dont la vérification de la consistance est

NP-complète. Le cas d'utilisation présenté est intéressant, car, pour la première fois dans la littérature, il est de taille conséquente. Ils indiquent qu'il existe 5323 features dans le noyau Linux. Vu que, dans nos tests présentés la Section 8.3.3, nous arrivons a convertir un FM de 13680 valeurs avec 2532 features différentes, ce qui est du même ordre de grandeur, nous pouvons conclure que notre approche est performante et utilisable pour des cas d'utilisation dans l'industrie.

9.2 Veille technologique

Notre implémentation repose sur FAMILIAR (décrit à la Section 2.3.2), qui lui-même utilise un grand nombre de projets connexes tels que FeatureIDE, S.P.L.A.R., SAT4J et JavaBDD ainsi que certains algorithmes comme celui de Czarnecki [25] qui reconstruit une hiérarchie à partir d'une expression logique. Une veille technologique soutenue à propos de ceux-ci est indispensable car des contributions pourraient apporter de grands avancements pour le processus d'extraction. Par exemple, si l'algorithme de Czarnecki monte en charge pour plus de 400 features, nous pourrions considérer l'utiliser pour de grands FMs. Cela permettrait d'obtenir des FMs de meilleure qualité vu que celui-ci reconstitue les *groupes Ou*, contrairement à notre stratégie d'optimisation par approximation de structure. Des informations complémentaires sur les limites actuelles de notre approche sont disponibles à la Section 8.4. De la même manière, si la librairie SAT4J améliorait l'heuristique de résolution des problèmes SAT, la performance de notre programme en serait favorablement impactée.

9.3 Travaux connexes

Une approche intéressante est décrite dans [35]. Ils extraient un diagramme de classe à partir de feuilles de calcul selon un processus automatisé. L'algorithme développé essaye de repérer des motifs types en parcourant les tableaux à deux dimensions des feuilles de calcul. Les motifs sont utilisés pour donner une interprétation en terme de classes et de relations à des données sémantiquement pauvres. Nos recherches respectives se ressemblent, car elles visent a extraire un modèle expressif à partir de données semi-structurées. Cependant, les réalisations sont fort différentes. En effet, ils essayent d'extraire automatiquement les données sans aucune intervention de l'utilisateur, ce qui mérite d'être étudié. Cette extraction amène néanmoins à un problème de validité des modèles générés. Les tests qu'ils ont dirigés ont montré que 40% des feuilles de calcul converties correspondent aux modèles cherchés. Notre approche, par contre, fait certaines interprétations, mais l'utilisateur peut paramétriser le processus d'extraction à tout niveau, ce qui lui permet d'obtenir le FM recherché. Il pourrait toutefois être intéressant de suivre

certaines pistes pour pousser l'automatisation de notre approche.

L'approche présentée dans [36] donne un processus d'extraction d'un FM à partir de données peu structurées diffusées sur le site *Softpedia.com*. Un système d'analyse de recommandation y est adjoint pour permettre à un utilisateur de sélectionner le produit qui lui correspond en donnant au système une description en langage naturel du produit qu'il veut obtenir. Cette approche utilise des algorithmes d'exploration textuelle et de groupement pour générer un feature model probabiliste et ensuite utiliser des algorithmes d'exploration des règles d'associations et des k-Nearest-Neighbor pour générer des recommandations sur des features particulières. Plusieurs points sont à étudier : (1) comparer leur processus d'extraction et celui présenté dans ce mémoire en vue de les améliorer de manière réciproque ; (2) étudier s'il est possible d'adjoindre leur système de recommandation à notre approche ; (3) étudier en quoi l'analyse du langage naturel permettrait d'améliorer notre processus d'extraction.

Notre processus d'extraction se base sur un modèle de conversion pour normaliser les données en entrée. Ce modèle crée des points pivots pour la fusion. Il n'y a donc pas de problèmes d'alignements dans notre approche. Il peut toutefois en apparaître si l'utilisateur fusionne deux FMs extraits à partir de catalogues de produits hétérogènes. Il devra alors les manipuler avec les opérateurs fournis par FAMILIAR pour adapter ces FMs et créer les pivots nécessaires à la fusion. Le lecteur qui est intéressé par les problèmes d'alignement, peut lire [28].

Employer des FMs probabilistes [21] conjointement avec notre processus d'extraction pour aider l'utilisateur dans le processus de configuration serait une extension pratique à notre approche. Les différentes probabilités associées aux nœuds pourraient être calculées à partir de métriques portant sur le catalogue de produit en entrée. Par exemple, si une valeur textuelle associée à une caractéristique particulière revient dans beaucoup de produits, nous pouvons lui donner une probabilité de sélection plus importante.

Chapitre 10

Conclusion

Les Feature Models (FMs) sont couramment utilisés dans l'ingénierie des lignes de produits logiciels (Software Product Lines - SPLs) pour documenter et raisonner sur la variabilité relative à une famille de produits. Les produits sont alors décrits en terme de caractéristiques (appelées features), structurées hiérarchiquement, qui peuvent être obligatoire ou optionnelles, représenter des alternatives, ou dépendre d'autres caractéristiques.

La modélisation manuelle d'un FM constitue une lourde tâche, consommatrice en temps et sujette à erreurs. À partir d'une certaine taille, il peut nécessiter des contraintes complexes et non intuitives. Il est donc irréalisable de construire un FM à partir d'un cas industriel comportant plusieurs milliers de features. Généralement, dans le cas de la modélisation d'une SPL déjà existante, la variabilité des produits est déjà documentée dans des catalogues qui consistent souvent en des fichiers peu structurés qui caractérisent les produits selon différentes perspectives. Dans ce mémoire, nous proposons une procédure automatisée et générique qui extrait un FM compact et représentatif d'un catalogue de produits existant. Notre approche vise à permettre à des organisations de convertir leurs SPLs existantes en des FMs avec très peu d'investissement.

Pour aider l'utilisateur à paramétriser le processus d'extraction, nous avons développé un langage déclaratif dédié. Il permet de filtrer les données en entrée, de définir leur interprétation en terme de variabilité et de structurer la hiérarchie du FM final. Une fois les directives écrites, notre processus va les transformer, ainsi que les données entrées, en une représentation logique. L'idée clé de notre approche est de dériver une formule propositionnelle qui « imite » toutes les combinaisons valides des caractéristiques de produits et qui peut être analysée par les solveurs SAT ou de diagrammes de décision binaire. Cette formule est définie sur un ensemble de variables booléennes dont chacune correspond à une caractéristique de produit. Grâce un algo-

rithme de fusion, nous obtenons une représentation logique et compacte de l'entièreté de la variabilité du catalogue de produits donné en entrée. Nous créons alors, à partir de cette formule, une hiérarchie de feature que nous appelons FM final.

Nos résultats d'évaluation démontrent la scalabilité et la praticabilité de l'approche sur des données publiques et larges ainsi que la validité des FMs produits. La montée en charge de notre application a été rendue possible par l'implémentation de diverses optimisations, dont la création d'un algorithme de fusion dédié à l'extraction de la variabilité. Par souci de généricité de notre approche, nous proposons une série d'extensions et nous avons implémenté un pont entre FAMILIAR et TVL, deux formalismes de représentation de FM. Un effort particulier a été fourni à l'implémentation des diverses contributions pour qu'elles soient claires, extensibles et qu'elles comportent peu de bugs.

Nous imaginons une série de perspectives dans le travail sur notre approche. Elles sont liées aux limitations de notre processus. Il pourrait être intéressant (1) d'étendre le processus d'extraction pour que celui-ci puisse convertir une base de données relationnelle ou d'autres formalismes structurés, et ce, de façon générique ; (2) de créer un mode d'extraction qui fait des interprétations supplémentaires sur le contenu du catalogue de produits pour que l'utilisateur n'ait pas a paramétrer le processus. Le processus serait alors entièrement automatique ; (3) de trouver une architecture permettant de définir facilement un nouvel accesseur aux données et de l'intégrer directement dans le DSL pour réduire l'effort de l'utilisateur ; (4) de faire en sorte que notre optimisation par structure approximée représente des *groupes Ou* pour rendre les FMs extraits plus expressifs ; (5) d'étudier en profondeur l'activité de filtrage (scoping) et améliorer l'outil existant. L'extension de cette activité permettrait une sélection plus fine des données en entrée. Des projets fort avancés sur le sujet existent déjà comme Google Refine [32].

Pour que notre travail soit réutilisé à plus large échelle, nous pensons qu'il faudrait créer un package tout-en-un comportant des interfaces permettant (1) d'importer différents formats de fichier ; (2) de manipuler et filtrer ces données ; (3) de paramétriser le processus d'extraction ; (4) de lancer la conversion et de récupérer le FM final dans divers formalismes de sorties comme FAMILIAR ou TVL ; (5) de visualiser la hiérarchie du FM final et de le configurer. En effet, le cadre d'utilisation actuel de notre approche est expérimental. Il nécessite l'installation d'un grand nombre de librairies, de plugins et reste cantonné à eclipse. Un point positif soulevé est que le processus d'extraction est scalable pour un grand nombre de features. L'utilisateur pourra donc charger sans problème un catalogue de produits issus de l'industrie.

Bibliographie

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Composing Feature Models. In *2nd Int'l Conference on Software Language Engineering (SLE'09)*, LNCS, page 20, 2009.

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Comparing Approaches to Implement Feature Model Composition. In *6th European Conference on Modelling Foundations and Applications (ECMFA)*, volume 6138 of *LNCS*, page 16, June 2010.

[3] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Managing Multiple Feature Models Using Merging Techniques. *Journal of Object Technology (ETH Zurich)*, October 2010. submitted : currently under review.

[4] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. A Domain-Specific Language for Managing Feature Models. In *Symposium on Applied Computing (SAC)*, , Taiwan, March 2011. Programming Languages Track, ACM.

[5] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Managing Feature Models with FAMILIAR : a Demonstration of the Language and its Tool Support. In *Fifth International Workshop on Variability Modelling of Software-intensive Systems(VaMoS'11) AR=55%*, VaMoS, Namur, Belgium, January 2011. ACM.

[6] Mathieu Acher, Philippe Collet, Philippe Lahire, Sabine Moisan, and Jean-Paul Rigault. Modeling Variability from Requirements to Runtime. In *16th International Conference on Engineering of Complex Computer Systems(ICECCS'11) AR=31%*, , Las Vegas, April 2011. IEEE.

[7] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5) :49–84, July/August 2009.

[8] A Text based Variability Language. `http://www.info.fundp.ac.be/~acs/tvl/`.

[9] Don S. Batory. Feature models, grammars, and propositional formulas. In *SPLC '05*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

[10] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. *LNCS, Advanced Information Systems Engineering : 17th International Conference, CAiSE 2005*, 3520 :491–503, 2005.

[11] David Benavides, Sergio Segura, and Antonio Ruiz-Cortes. Automated Analysis of Feature Models 20 years Later : a Literature Review. *Information Systems, Elsevier*, 2010.

[12] Thorsten Berger, Steven She, Rafael Lotufo, Andrezj Wasowski, and Krzysztof Czarnecki. Variability modeling in the real : A perspective from the operating systems domain. In *25th IEEE/ACM International Conference on Automated Software Engineering*, 09/2010 2010.

[13] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. Introducing TVL, a text-based feature modelling language. In *VaMoS'10*, pages 159–162, 2010.

[14] Peter Buneman. Semistructured data. *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems - PODS '97*, pages 117–121, 1997.

[15] Fei Cao, Barrett R. Bryant, Carol C. Burt, Zhisheng Huang, Rajeev R. Raje, Andrew M. Olson, and Mikhail Auguston. Automating feature-oriented domain analysis. In Ban Al-Ani, Hamid R. Arabnia, and Youngsong Mun, editors, *Software Engineering Research and Practice*, pages 944–949. CSREA Press, 2003.

[16] Andreas Classen, Quentin Boucher, Paul Faber, and Patrick Heymans. The TVL specification. Technical Report P-CS-TR SPLBT-00000003, PReCISE Research Center, University of Namur, Namur, Belgium, 2010.

[17] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling : Syntax and semantics of TVL. *Science of Computer Programming, Special Issue on Software Evolution*, 2010.

[18] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. *What's in a Feature : A Requirements Engineering Perspective*, pages 16–30. Fundamental Approaches to Software Engineering (FASE), 2008.

[19] Paul Clements and Linda M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 2001.

[20] CVS. `http://cvs.nongnu.org/`.

[21] K. Czarnecki, S. She, and A. Wasowski. Sample spaces and feature models : There and back again. In *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 22–31, 2008.

[22] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. In *Software Process Improvement and Practice*, pages 7–29, 2005.

[23] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration through Specialization and Multilevel Configuration of Feature Models. *Software Process : Improvement and Practice*, 10(2) :143–169, 2005.

[24] Krzysztof Czarnecki, Chang Hwan Peter Kim, and Karl Trygve Kalleberg. Feature models are views on ontologies. In *SPLC '06 : Proceedings of the 10th International on Software Product Line Conference*, pages 41–51. IEEE Computer Society, 2006.

[25] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics : There and back again. In *SPLC 2007*, pages 23–34, 2007.

[26] Merijn de Jonge and Joost Visser. Grammars as feature diagrams. In *Proceedings of Workshop on Generative Programming*, pages 23–24, 2002.

[27] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Product derivation in software product families : a case study. *Journal of Systems and Software*, 74(2) :173–194, 2005.

[28] Jérôme Euzenat and Pavel Shvaiko. *Ontology Matching*. 2007.

[29] FaMa. `http://www.isa.us.es/fama/`.

[30] FAMILIAR : FeAture Model scrIpt Language for manIpulation and Automatic Reasonning. `http://nyx.unice.fr/projects/familiar/`.

[31] Git. `http://git-scm.com/`.

[32] Google Refine. `http://code.google.com/p/google-refine/`.

[33] Florian Heidenreich, Pablo Sanchez, Joao Santos, Steffen Zschaler, Mauricio Alferez, Joao Araujo, Lidia Fuentes, Uira Kulesza amd Ana Moreira, and Awais Rashid. Relating feature models to other models of a software product line : A comparative study of featuremapper and vml*. *Transactions on Aspect-Oriented Software Development VII, Special Issue on A Common Case Study for Aspect-Oriented Modeling*, 6210 :69–114, 2010.

[34] Felienne Hermans and Martin Pinzger. Automatically Extracting Class Diagrams from Spreadsheets. *ECOOP 2010 Object-Oriented*, 2010.

[35] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Automatically extracting class diagrams from spreadsheets. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 52–75, Berlin, Heidelberg, 2010. Springer-Verlag.

[36] Negar Hariri Jane Cleland-Huang Bamshad Mobasher Carlos Castro-Herrera Mehdi Mirakhorli Horatiu Dumitru, Marek Gibiec. On-demand feature recommendations derived from mining public product descriptions. IEEE International Conference on Software Engineering, 2011.

[37] Arnaud Hubaux, Quentin Boucher, Herman Hartmann, Raphaël Michel, and Patrick Heymans. Evaluating a textual feature modelling language : four industrial case studies. In *Proceedings of the Third international conference on Software language engineering*, SLE'10, pages 337–356, Berlin, Heidelberg, 2011. Springer-Verlag.

[38] Mikolás Janota, Victoria Kuzina, and Andrzej Wasowski. Model construction with external constraints : An interactive journey from semantics to syntax. In *MoDELS'08*, volume 5301 of *LNCS*, pages 431–445. Springer, 2008.

[39] JavaBDD. `http://javabdd.sourceforge.net/index.html`.

[40] Isabel John. Capturing product line information from legacy user documentation. In Timo Käköla and Juan Carlos Duenas, editors, *Software Product Lines*, pages 127–159. Springer Berlin Heidelberg, 2006.

[41] Isabel John and Michael Eisenbarth. A decade of scoping : a survey. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 31–40, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[42] Junit. `http://www.junit.org/`.

[43] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, 1990.

[44] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Gerard Jounghyun Kim, and Euiseob Shin. Form : A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5 :143–168, 1998.

[45] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. FeatureIDE : Tool framework for feature-oriented software development. In *Proceedings of the 31th International Conference on Software Engineering (ICSE)*, pages 611–614, Los Alamitos, CA, USA, May 2009. IEEE Computer Society. Formal Demonstration paper, Acceptance rate : 33 % (24/72).

[46] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0053381.

[47] Marcilio Mendonca, Moises Branco, and Donald Cowan. S.p.l.o.t. : software product lines online tools. In *OOPSLA '09 : Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762, New York, NY, USA, 2009. ACM.

[48] Marcilio Mendonca, Andrzej Wasowski, Krzysztof Czarnecki, and Donald Cowan. Efficient compilation techniques for large scale feature models. In *GPCE '08*, pages 13–22. ACM, 2008.

[49] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37 :316–344, December 2005.

[50] Andreas Metzger, Klaus Pohl, Patrick Heymans, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the documentation of variability in software product lines : A separation of concerns, formalization and automated analysis. In *Requirements Engineering (RE '07)*, pages 243–253, 2007.

[51] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. *SIGSOFT Softw. Eng. Notes*, 29 :127–136, October 2004.

[52] An SPL of SPL Techniques and Tools. `http://download.lero.ie/spl/s2t2/`.

[53] SAT4J. `http://www.sat4j.org/`.

[54] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Comput. Netw.*, 51(2) :456–479, 2007.

[55] S. Segura, D. Benavides, A. Ruiz-Cortes, and P. Trinidad. Automated merging of feature models using graph transformations. *Post-proceedings of the Second Summer School on GTTSE*, 5235 :489–505, 2008.

[56] Y. Shafranovich. RFC 4180-Common Format and MIME Type for Comma-Separated Values (CSV) Files. *The International Society*, 54 :258, 2005.

[57] Subversion. `http://subversion.tigris.org/`.

[58] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning about edits to feature models. In *ICSE'09*, pages 254–264. IEEE, 2009.

[59] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`.

[60] Software Product Line Online Tools. `http://www.splot-research.org/`.

[61] `http://www.wikimatrix.org/`.

[62] XML. `http://www.w3.org/TR/2008/REC-xml-20081126/`.

[63] Xtext. `http://www.eclipse.org/Xtext/`.

[64] Zest. `http://www.eclipse.org/gef/zest/`.